

# SuperSPARC & MultiCache Controller User's Manual

**SPARC** *Technology Business*



A Sun Microsystems, Inc Business  
2550 Garcia Avenue, Mountain View, CA U.S.A. 94043  
(415) 336-2801

Revision 1.0 - April 1994 - Preliminary





Products Rights Notice:

Copyright © 1991-2008 Sun Microsystems, Inc. 4150 Network Circle, Santa Clara, California 95054, U.S.A. All Rights Reserved

You understand that these materials were not prepared for public release and you assume all risks in using these materials. These risks include, but are not limited to errors, inaccuracies, incompleteness and the possibility that these materials infringe or misappropriate the intellectual property right of others. You agree to assume all such risks.

THESE MATERIALS ARE PROVIDED BY THE COPYRIGHT HOLDERS AND OTHER CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS (INCLUDING ANY OF OWNER'S PARTNERS, VENDORS AND LICENSORS) BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THESE MATERIALS, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Sun, Sun Microsystems, the Sun logo, Solaris, OpenSPARC T1, OpenSPARC T2 and UltraSPARC are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon architecture developed by Sun Microsystems, Inc. UNIX is a registered trademark in the U.S. and other countries, exclusively licensed through X/Open Company, Ltd. The Adobe logo is a registered trademark of Adobe Systems, Incorporated. Part of the products covered by these materials may be derived from the Berkeley BSD systems licensed by the University of California. Sun Microsystems, Inc. has intellectual property rights relating to technology embodied in the product described in these materials. This distribution may include materials developed by third parties who have intellectual property rights therein. Products covered by and information contained in these materials may be controlled by U.S. Export Control laws and may be subject to the export or import laws in other countries. Nuclear, missile, chemical biological weapons or nuclear maritime end uses or end users, whether direct or indirect, are strictly prohibited. Export or reexport to countries subject to U.S. embargo or to entities identified on U.S. export exclusion lists, including, but not limited to, the denied persons and specially designated nationals lists may be prohibited.

# *SuperSPARC Family STP1020 & STP1090 Series User's Manual*

This manual is the Beta release of the SuperSPARC User's Guide. This is a preliminary version. While this version of the User's Guide represents what we believe to be correct data, the information contained in this manual is subject to change without notice.

This manual is applicable to all revisions of the SuperSPARC microprocessor and the MultiCache Controller chips. Some features might be specifically available on certain revisions only. For revision-specific information, please see Appendix E and Appendix F.

Revision 1.0  
April 1994



## Preface

# Read This First

---

This preface summarizes the chapters, lists related documentation, and describes the style and symbol conventions used in this manual.

### *How This Manual Is Organized*

This document contains the following chapters:

- |                  |  |
|------------------|--|
| <b>Chapter 1</b> | <b>Introduction</b><br>An introduction to SuperSPARC based systems   |
| <b>Chapter 2</b> | <b>Summary of the SPARC Architecture</b><br>Summarizes the major features of the SPARC Version 8 architecture.   |
| <b>Chapter 3</b> | <b>SuperSPARC Introduction</b><br>Contains an introduction to the SuperSPARC processor, a highly integrated high-performance implementation of the SPARC architecture. |
| <b>Chapter 4</b> | <b>Register Summary</b><br>Summarizes the registers in the SuperSPARC processor's integer and floating point units.  |
| <b>Chapter 5</b> | <b>Principles of Operation</b><br>Describes the SuperSPARC processor's pipeline operation and how to use it efficiently.   |
| <b>Chapter 6</b> | <b>Code-Generation Principles</b><br>Presents guidelines for increasing the performance of programs on the SuperSPARC processor.                                       |
| <b>Chapter 7</b> | <b>Instructions</b><br>Outlines the instructions that expand upon the SPARC Version 8 instruction set.   |
| <b>Chapter 8</b> | <b>Memory Model</b><br>Describes the memory in a SuperSPARC-based system.  |
| <b>Chapter 9</b> | <b>MMU Operation</b><br>Describes the SuperSPARC processor's MMU, which is compatible with the SPARC Reference MMU Specification.                                      |

- Chapter 10 Caches/Store Buffer**  
Describes SuperSPARC's internal caches and store buffer.
- Chapter 11 Floating-Point Unit Operation**  
Describes the floating-point operations, concentrating on the floating-point-queue interface, special numeric cases, and floating-point exceptions.
- Chapter 12 Traps**  
Gives an overall view of the types of traps that may occur during normal or exceptional operation of SuperSPARC.
- Chapter 13 Reset**  
Describes how SuperSPARC implements various forms of reset.
- Chapter 14 Startup Procedure**  
Gives an example of code from a reset handler. The example shows critical initializations required for proper operation of the SuperSPARC processor.
- Chapter 15 Diagnostic Operation**  
Details software debugging capabilities and external monitors.
- Chapter 16 MultiCache Controller (MXCC)**  
Describes the MultiCache Controller, an optional external cache controller for SuperSPARC processors.
- Chapter 17 MBus**  
Details the operation of the SPARC MBus on the SSP and the MXCC. MBus is a high-speed interface that connects SPARC processor modules to physical memory and I/O modules.
- Chapter 18 VBus**  
Details the operation of the VBus interface used between SuperSPARC and the MXCC.
- Chapter 19 XBus**  
Details the operation of the XBus packet bus interface used for multiprocessor SuperSPARC systems.
- Chapter 20 BootBus**  
Describes the BootBus, a simple synchronous 12-pin interface provided by the MXCC for accessing an EPROM for bootstrap loading and for accessing other low-speed peripherals.
- Chapter 21 JTAG Serial Scan Interface**  
Explains how the IEEE 1149.1 JTAG serial scan interface mechanism supports observation and control of the SuperSPARC processor for different applications.
- Chapter 22 Scan-Based Debug**  
Explains the use of the IEEE 1149.1 JTAG serial scan interface for software debugging.

- Chapter 23 Clocking**  
Describes SuperSPARC and MXCC essential clock requirements.
- Chapter 24 MBus Module**  
Describes an example application of a plug-in MBus module.
- Appendix A Instruction Summary**  
Contains a summary of SuperSPARC's instruction set.
- Appendix B ASI/Diagnostic Access**  
Provides a table of the SuperSPARC ASI assignments.
- Appendix C SuperSPARC Processor Pin Description Tables**  
Contains tables describing each pin on the SSP.
- Appendix D MultiCache Controller Pin Description Tables**  
Contains tables describing each pin on the MXCC.
- Appendix E SuperSPARC Revision Summary**  
Contains table describing salient features of SuperSPARC revisions.
- Appendix F MultiCache Controller Revision Summary**  
Contains table describing salient features of MultiCache Controller revisions.
- Appendix G Glossary**  
Contains a glossary of important terms and acronyms used in this book.

### ***Related Documentation***

The following related documents are also available.

- ☐ SuperSPARC (STP1020N, STP1020, STP1020A) Data Sheet
- ☐ MultiCache Controller (STP1090, STP1090A) Data Sheet
- ☐ The SPARC Architecture Manual, Version 8
- ☐ SPARC MBus Interface Specification

### ***Bibliography***

### ***Style and Symbol Conventions***

This document uses the following conventions:

- ☐ Program listings, program examples, and interactive displays are shown in a special font. Examples use a bold version of the special font for emphasis. Here is a sample program listing:

```
add %l0, %l1, %l2
sub %o1, %o2, %o3
and %o3, %o4, %o5
ld [%l2 + 0x10], %l3
```

- ☐ Hexadecimal Numbers will be written in the 0xnnn form. Thus 31 decimal written as in hexadecimal will be written 0x1F.
- ☐ Register fields will be shown in the form REGISTER.FIELD. For example, the Current Window Pointer field of the Processor Status Register will be written as PSR.CWP .
- ☐ Register fields named in ALL CAPITALS are both readable and writable by software. While register fields named in small letters are readable by software but are set by hardware (writing to these fields will not necessarily change their value). For example, TBR.TBA is set by the system software while TBR.tt is set by hardware on receipt of a trap.
- ☐ Register Conventions. The following are the conventions used in naming specific register fields.

**reserved, res, r** reserved for definition in future versions of the SPARC architecture. A reserved field should always be written to as zeros by software, but software should **not** assume a reserved field will read as zeros.

**unused, u** used to describe a register field that is not currently defined by the architecture. An unused field should always be written to as zeros by software, but when read an unused field will return an undefined value.

**zero, 0** used to describe a register field that will always read as zero. A write of any other value to a zero field will have no effect on a subsequent read, which will return zero.

## *Style and Symbol Conventions*

Following are other symbols and abbreviations used throughout this document.

Symbol	Definition	Symbol	Definition
ALU	Arithmetic Logic Unit	nPC	next Program Counter
ASI	Address Space Indicator	PA	Physical Address
ASR	Ancillary State Register	PC	Program Counter
BIST	Built-in Self-test	PIL	Processor Interrupt Level
CPI	Cycles Per Instruction	PLL	Phase-Locked Loop
CWP	Current Window Pointer	POST	Power-On Self-Test
EPROM	Erasable Programmable Read-only Memory	PPN	Physical Page Number
FIFO	First-in, First-out	PSO	Partial Store Ordering
FPU	Floating-point Unit	PSR	Processor State Register
IPC	Instructions Per Cycle	PTE	Page Table Entry
IQ	Instruction Queue	PTP	Page Table Pointer
IU	Integer Unit	RISC	Reduced Instruction Set Computer
JTAG	Joint Test Access Group (IEEE 1149.1)	SO	Sequential Ordering
LSB	Least Significant Byte (or Bit)	SSP	SuperSPARC Processor
MXCC	MultiCache Controller	TAP	Test Access Port
MMU	Memory Management Unit	TLB	Translation Lookaside Buffer
MOESI	Modified, Owned, Exclusive, Shared, Invalid	TSO	Total Store Ordering
MSB	Most Significant Byte (or Bit)	VA	Virtual Address
NaN	Not a Number	VPN	Virtual Page Number
NOP	No Operation	WIM	Window Invalid Mask

**Trademarks**

SBus, Sun-3, Sun-4, SunView, SunWindows, and Sun Workstation are trademarks of Sun Microsystems, Inc.

UNIX is a registered trademark of UNIX Systems Laboratories, Inc.

SPARC, MBus is a registered trademark of SPARC International.





# Introduction

---

---

---

The Sparc Technology Business SuperSPARC chipset comprises the SuperSPARC processor (SSP) and the optional MultiCache Controller (MXCC). This chapter describes the key components, features, and configurations of the chipset.

Topic	Page
1.1 Introduction to SuperSPARC Chipset .....	1-2
1.2 System Configurations .....	1-4

## 1.1 Introduction to SuperSPARC Chipset

The SuperSPARC chipset, which comprises the SuperSPARC advanced microprocessor and the MultiCache Controller, combines STB's strengths in high-performance semiconductor processes and Sun Microsystems' expertise in sophisticated computer systems.

### 1.1.1 Key Components

#### *SuperSPARC Microprocessor*

The processor's 3.1 million transistors form a highly integrated processing subsystem containing:

- ☐ Superscalar integer unit.
- ☐ Double-precision IEEE floating-point unit.
- ☐ 20K-byte instruction cache.
- ☐ 16K-byte data cache.
- ☐ SPARC Reference memory management unit (MMU).
- ☐ Eight-entry store buffer.
- ☐ Dual-mode bus interface.
- ☐ IEEE 1149.1 JTAG test and debug.

#### *SuperSPARC MultiCache Controller*

The MXCC is an optional external cache controller for the SuperSPARC processor (SSP). It is employed when a large secondary cache or an interface to a non-MBus system is a requirement. The MXCC contains:

- ☐ Cache tags and control for .5 to 2M bytes of external cache memory.
- ☐ Synchronization logic to allow the processor to be clocked faster than the system bus.
- ☐ Block copy/fill logic.
- ☐ Dual-bus interface:
  - MBus (CMOS).
  - XBus (GTL).

### **1.1.2 Features**

The chipset's features aid in building a wide range of high-performance workstations and server computers.

☐ **SPARC Compatibility**

The SSP is fully compatible with the SPARC Architecture, version 8, from SPARC International. Compatibility ensures that thousands of SPARC application programs run on SuperSPARC-based systems.

☐ **Superscalar Execution**

The processor can execute up to three instructions per clock cycle. This internal parallelism accelerates all applications while allowing system clock rates to remain manageable. Even programs that have not been recompiled for superscalar execution run faster on SuperSPARC.

☐ **Built-in MBus Multiprocessing**

The chipset supports the SPARC standard MBus directly and, in several configurations, connects directly to MBus without any glue logic. Each chip contains all logic necessary for shared memory multiprocessing with fully coherent caches on MBus.

☐ **High Integration**

The SuperSPARC chipset simplifies system design by integrating the components of a high-performance processing subsystem onto very few chips.

☐ **Multiple Configurations**

The processor can be used alone or with MXCC and cache RAMs. The chipset can be used in uniprocessor or multiprocessor systems and supports MBus systems and systems based on customer-defined buses.

## 1.2 System Configurations

The chipset supports the configurations shown in Table 1-1. This section describes stand-alone and high-performance MBus and XBus configurations. The minimum MBus and XBus configurations offer few advantages.

Table 1-1. SuperSPARC Chipset Configurations

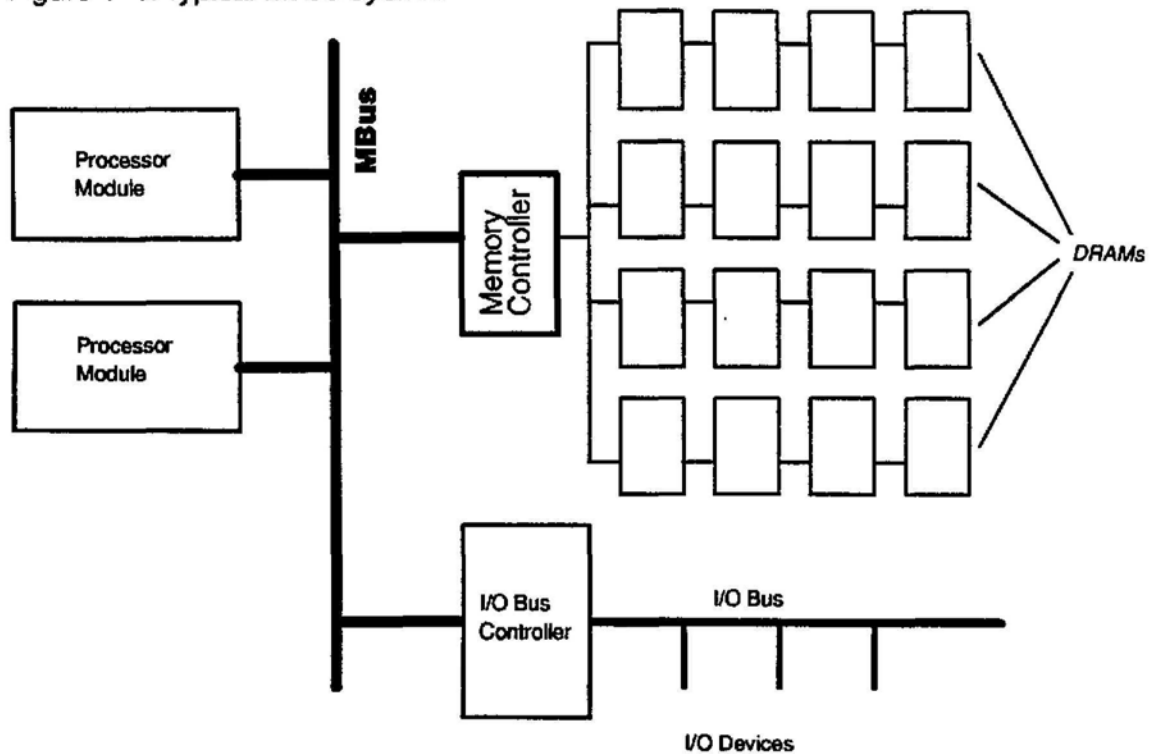
Configuration	SuperSPARC	MXC C	SRAMs	Bus
Stand-Alone	√			MBus and VBus
Minimum MBus	√	√		MBus
Minimum XBus	√	√		customer-defined
High-Performance MBus	√	√	√	MBus
High-Performance XBus	√	√	√	customer-defined

### 1.2.1 MBus Configurations

MBus is a SPARC standard that connects high-performance processors to memory and maintains coherent memory and caches in shared memory multi-processors. Figure 1-1 shows a typical MBus system.

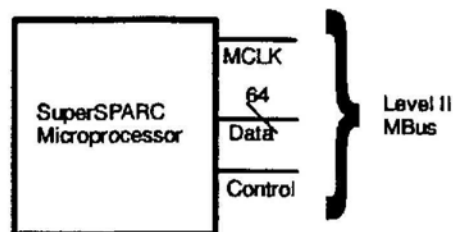
Both the stand-alone and high-performance MBus configurations are available as SPARC-standard MBus plug-in modules. The modules allow plug-in performance upgrades.

Figure 1-1. Typical MBus System



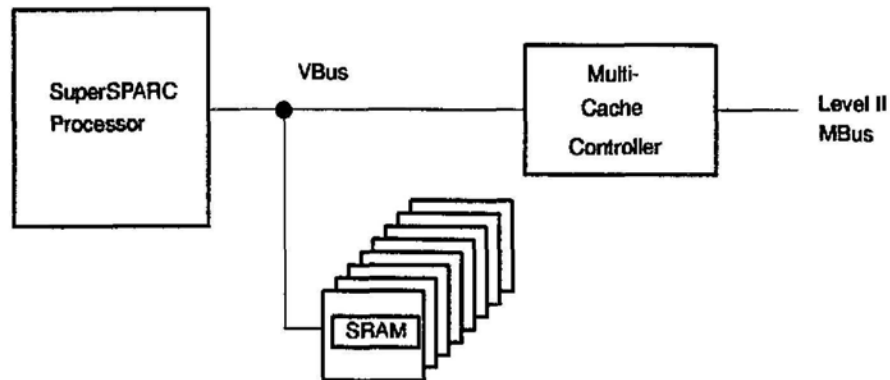
In stand-alone configurations, the processor connects directly to the MBus, as shown in Figure 1-2. Since this configuration does not use MXCC or cache RAM, it is cost-effective. For most applications, using more than two processors will saturate the MBus bandwidth. MBus supplies the processor clock.

Figure 1-2. Stand-Alone Configuration



The high-performance configuration is diagrammed in Figure 1-3. The external cache memory provides significant performance improvement and greatly decreases bus traffic in order to support more processors on a system bus. The SRAMs are industry-standard 128K-bit x 9 synchronous SRAMs. The MXCC allows the processor to be clocked faster than the system bus.

Figure 1-3. High-Performance Configuration



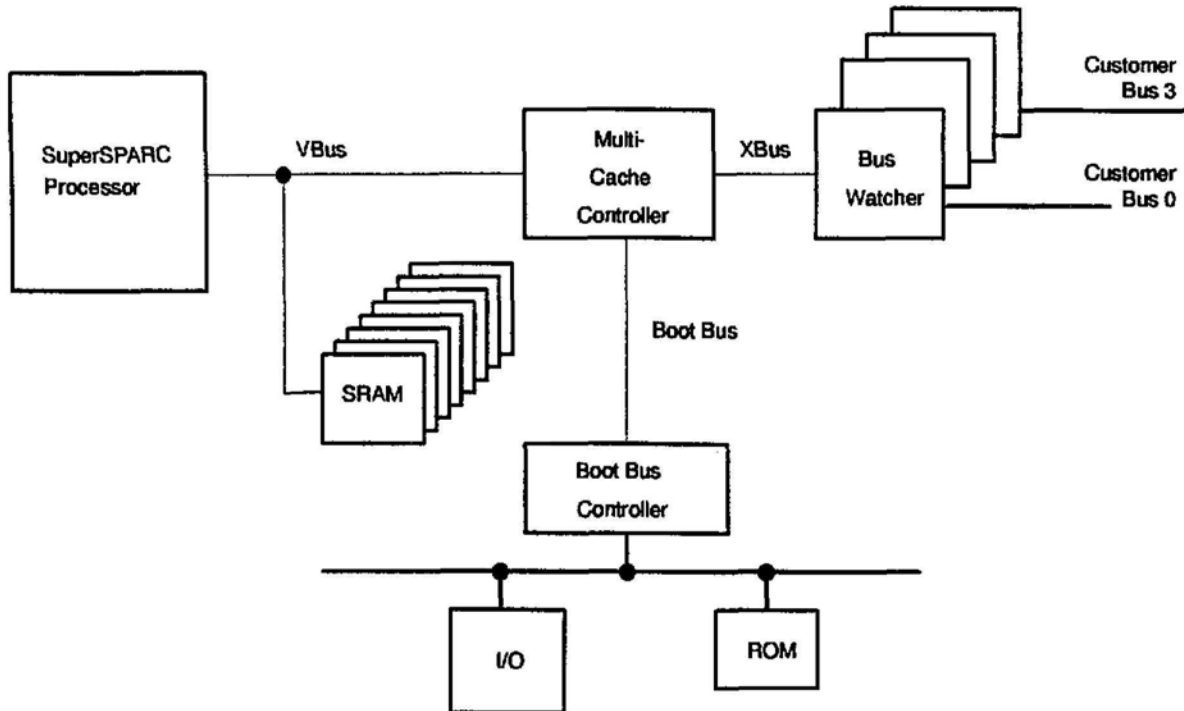
### 1.2.2 XBus Configurations

To support systems that do not use MBus, MXCC supports external system bus interfaces—or bus watchers (BWs)—in its XBus configurations. The BW interfaces MXCC to a particular system bus. Figure 1-4 shows the High-Performance XBus configuration.

The synchronous packet-switched XBus was developed by Xerox PARC and extends the functions of MXCC to a BW. The bus uses GTL, a low-voltage electrical interface, for the high speed and noise immunity required in massively parallel configurations.

MXCC supports up to four external BWs. Each BW interfaces with a system bus so that up to four system buses can be used to increase available bandwidth between processors and memory. Multiple system buses can support a large number of SuperSPARC processors—up to 64 for some applications.

Figure 1-4. XBus System with External Bus Watchers







## Summary of the SPARC Architecture

---

---

The SuperSPARC processor (SSP) is an implementation of *The SPARC Architecture, version 8*. This chapter summarizes the major features of the SPARC architecture; for a complete description, consult *The SPARC Architecture Manual, version 8*.

Topic	Page
2.1 Introduction .....	2-2
2.2 Instructions .....	2-5
2.3 Memory Model .....	2-9
2.4 Input/Output .....	2-10
2.5 Traps .....	2-11
2.6 SPARC Reference MMU .....	2-13

## 2.1 Introduction

The SSP implements the SPARC Architecture, version 8, and a SPARC Reference Memory Management Unit (MMU) as specified in *The SPARC Architecture Manual*. SPARC is a standard, and, while SuperSPARC meets the SPARC standard, it does not implement all of the optional features (e.g., quadruple-precision floating-point).

The material in this chapter is adapted from *The SPARC Architecture Manual*, copyright 1991 by SPARC International, Inc., based on technology developed by Sun Microsystems, Inc. Used by permission.

These are the principal features of SPARC, version 8:

- ☐ A linear, 32-bit address space.
- ☐ A small number of simple instruction formats.
- ☐ Load/store architecture (no memory/operate instructions).
- ☐ Three register addresses (two operands and a separate destination).
- ☐ Large-windowed register file.
- ☐ Separate floating-point register file.
- ☐ Delayed control transfer.
- ☐ Fast trap handling.
- ☐ Multiprocessor synchronization instructions.
- ☐ Tagged data instructions.
- ☐ Optional coprocessor support (not supported on SuperSPARC).
- ☐ Three multiprocessor memory models.

SPARC has an instruction set architecture with 32-bit integer and 32-, 64-, and 128-bit IEEE Standard 754 floating-point arithmetic as its principal data types. It defines general-purpose integer, floating-point, special state/status registers and 69 basic instruction operations. These instructions are all encoded in 32-bit-wide instruction formats. The load/store instructions address a linear,  $2^{32}$ -byte address space.

A SPARC processor is logically composed of the following:

- ☐ An integer unit (IU),
- ☐ A floating-point unit (FPU), and
- ☐ An optional coprocessor (CP), each with its own registers.

IU and FPU registers are 32 bits wide. Instruction operands are generally single registers or register pairs. The processor can be in either of two modes: user or supervisor. In supervisor mode, the processor can execute any instruction, including the privileged (supervisor-only) instructions. In user mode, an attempt to execute a privileged instruction will cause a trap to supervisor software. User-application programs execute while the processor is in user mode.

### 2.1.1 Integer Unit

The IU contains the general-purpose registers and controls the overall operation of the processor. The IU executes the integer arithmetic instructions and computes memory addresses for loads and stores. It also maintains the program counters and controls instruction execution for the FPU and the CP.

An implementation can contain from 40 to 520 general-purpose 32-bit *r* registers. This corresponds to a grouping of the registers into eight global *r* registers, plus a circular stack of from 2 to 32 sets of 16 registers each, known as register windows (see Section 4.1). Since the number of register windows present (NWINDOWS) is implementation-dependent, the total number of registers is also implementation-dependent.

SuperSPARC has eight register windows, for a total of 136 *r* registers.

An instruction can access the eight global registers and the current window of 24 *r* registers. A window of 24 registers is composed of a 16-register set—divided into eight in and eight local registers—together with the eight in registers of an adjacent register set, addressable from the current window as out registers. The current window is specified by the current window pointer (CWP) field in the processor state register (PSR). Window overflow and underflow are detected via the window invalid mask (WIM) register, which is controlled by supervisor software. The actual number of windows in a SPARC implementation is invisible to a user-application program.

When the IU accesses memory, the IU appends an address space identifier (ASI) to the address. The ASI encodes the address according to whether the processor is in supervisor or user mode and whether the access is to instruction memory or to data memory. Supervisor programs can make access to program-controlled address spaces by using the Load/Store ASI instructions.

### 2.1.2 Floating-Point Unit (FPU)

The FPU has 32 32-bit floating-point *f* registers. Double-precision values occupy an even-odd pair of registers. Thus, the floating-point registers can hold a maximum of 32 single-precision, 16 double-precision, or 8 quad-precision values.

Floating-point load/store instructions are used to move data between the FPU and memory. The memory address is calculated by the IU. Floating-point operate (FPop) instructions perform the actual floating-point arithmetic.

The floating-point data formats and instruction sets conform to the IEEE Standard for binary floating-point arithmetic, ANSI/IEEE 754-1985. However, SPARC does not require that all aspects of the standard, such as gradual underflow, be implemented in hardware. An alternate method of indicating that a floating-point instruction failed to produce a correct ANSI/IEEE Standard 754-1985 result is to generate a special floating-point unfinished or unimplemented exception. Software must emulate any functionality not present in the hardware. If an FPU is not present, or if the enable floating-point (PSR.EF) bit in the PSR is 0, an attempt to execute a floating-point instruction will generate an `illegal_instruction` trap. In either of these cases, software must emulate the trapped floating-point instruction.

SuperSPARC never generates a floating-point unfinished trap since it handles gradual underflow in hardware. SuperSPARC implements all 32-bit single-precision floating-point and 64-bit double-precision floating-point instructions. It implements no 128-bit quad-precision floating-point instructions. Quad-precision instructions trap with a floating-point unimplemented exception.

### **2.1.3 Coprocessor (CP)**

While the SPARC architecture supports an optional coprocessor, SuperSPARC contains no provisions for a coprocessor.

## 2.2 Instructions

Instructions fall into six basic categories:

- ☐ Load/store.
- ☐ Arithmetic/logical/shift.
- ☐ Control transfer.
- ☐ Read/write.
- ☐ Floating-point operate.
- ☐ Coprocessor operate.

### 2.2.1 Load/Store

Load/store instructions are the only instructions that access memory. They use two *r* registers or an *r* register and a signed 13-bit immediate value to calculate a 32-bit, byte-aligned memory address. The destination field of the load/store instruction specifies an *r* register, *f* register, or coprocessor register that supplies the data for a store or receives the data from a load.

The processor appends an ASI to every access. This ASI is derived in one of two ways: directly from a Load/Store ASI instruction, or from a default ASI that is generated based on user data access or supervisor data access. This ASI is used by the MMU and/or the system.

Integer load and store instructions support byte, half-word (16-bit), word (32-bit), and double-word (64-bit) accesses. Versions of integer load instructions perform sign-extension on 8- and 16-bit values as the values are loaded into the destination register. Floating-point and coprocessor load and store instructions support word and double-word memory accesses.

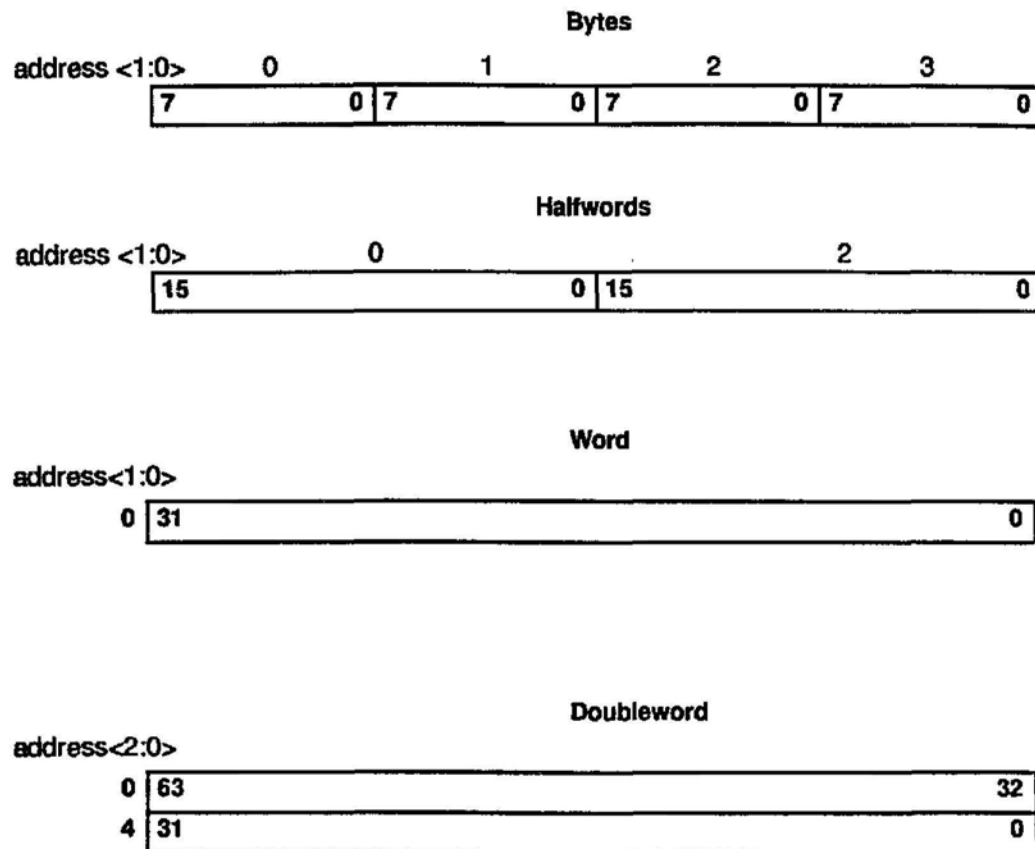
#### *Alignment Restrictions*

Half-word accesses must be aligned on two-byte boundaries, word accesses must be aligned on four-byte boundaries, and double-word accesses must be aligned on eight-byte boundaries. An improperly aligned address in a load or store instruction causes a trap.

#### *Addressing Conventions*

SPARC is a "big-endian" architecture: the address of a double-word, word, or half-word is the address of its most significant byte. Increasing the address means decreasing the significance of the unit being accessed. Addressing conventions are illustrated in Figure 2-1.

Figure 2-1. Addressing Conventions

**Load/Store Alternate**

Special, privileged versions of the load and store integer instructions (the load/store alternate instructions) can directly specify an arbitrary eight-bit address space identifier for the load/store data access. The privileged load/store alternate instructions can be used by supervisor software to access special protected registers, such as an MMU, cache-control, processor control registers, and other processor- or system-dependent values.

## Separate I&D Memories

Most specifications in *The SPARC Architecture Manual* are written as if the store instructions wrote to the same memory from which instructions were accessed. However, an implementation may explicitly partition instructions and data into independent instruction and data memories (caches), commonly referred to as a "Harvard" architecture or "split I & D caches." If a program includes self-modifying code, it must issue FLUSH instructions (or supervisor calls that have an equivalent effect) for the addresses to which new instructions are written. A FLUSH instruction ensures that the data previously written by a store instruction is seen by subsequent instruction fetches from the given address.

### 2.2.2 Arithmetic/Logical/Shift

The arithmetic/logical/shift instructions perform arithmetic, tagged-arithmetic, logical, and shift operations. With one exception, these instructions compute a result that is a function of the two source operands; the result is either written into a destination register or discarded. The exception is a specialized instruction, SETHI, which (along with a second instruction) can be used to create a 32-bit constant in an *r* register. Shift instructions can be used to shift the contents of an *r* register left or right by a given distance. The shift distance can be specified by a constant in the instruction or by the contents of an *r* register.

The integer multiply instructions perform a signed or unsigned  $32 \times 32 \rightarrow 64$ -bit multiplication. The integer division instructions perform a signed or unsigned  $64 \div 32 \rightarrow 32$ -bit division. Versions of multiply and divide set the condition codes. Division by zero causes a trap.

The tagged-arithmetic instructions assume that the least significant two bits of the operands are data-type "tags." When there is an arithmetic overflow or if any of the operands' tag bits are nonzero, these instructions set the overflow condition code bit. Some versions of tagged arithmetic instructions trap when either of these conditions occurs.

### 2.2.3 Control Transfer

Control-transfer instructions (CTIs) include program counter (PC)-relative branches and calls, register-indirect jumps, and conditional traps. Most of the control-transfer instructions are delayed control-transfer instructions (DCTIs), in which the instruction immediately following the DCTI is executed before the control transfer to the target address is completed.



The instruction following a delayed control-transfer instruction is called a delay instruction. The delay instruction is always fetched, even if the delayed control transfer is an unconditional branch. A bit in the delayed control-transfer instruction, however, can cause the delay instruction to be annulled (that is, to have no effect) if the branch is not taken (or, in the branch always case, if the branch is taken).

Branch and CALL instructions use PC-relative displacements. The jump and link (JML) instruction uses a register-indirect target address. It computes its target address as either the sum of two *r* registers or the sum of an *r* register and a 13-bit signed immediate value. The branch instruction provides a displacement of  $\pm 8\text{M}$  bytes, while the CALL instruction's 30-bit word displacement allows a control transfer to an arbitrary 32-bit instruction address.

Ticc instructions cause a non-delayed transfer to a trap table entry (see Section 2.5 and Chapter 12).

### 2.2.4 Read/Write

The read/write register instructions read and write the contents of software-visible state/status registers. Software can use read/write "ancillary state register" instructions to read/write unique implementation-dependent processor registers. Whether each of these instructions is privileged is implementation-dependent. (See Section 7.3, Write PSR.)

### 2.2.5 Floating-Point Operate

FPop instructions perform all floating-point calculations. They are register-to-register instructions that operate on the floating-point *f* registers. Like arithmetic/logical/shift instructions, an FPop computes a result that is a function of one or two source operands. Specific floating-point operations are selected by a subfield of the FPop1/FPop2 instruction formats. See Chapter 11.

### 2.2.6 Coprocessor Operate

Coprocessor operate (CPop) instructions are defined by the implemented coprocessor, if any. These instructions are specified by the CPop1 and CPop2 instruction formats.

## 2.3 Memory Model

The SPARC memory model has two functions:

- ☐ It defines the semantics of such memory operations as load and store, and
- ☐ It specifies the relationship between the order in which these operations are issued by a processor and the order in which they are executed by memory.

The model applies both to uniprocessors and shared-memory multiprocessors (see Chapter 8).

The standard memory model is called total store ordering (TSO). All SPARC implementations must provide at least TSO. An additional model called partial store ordering (PSO) is defined to allow higher-performance memory systems to be built. If present, this model is enabled via a mode bit—for example, in an MMU control register. Machines that implement strong consistency (also called strong ordering) automatically support both TSO and PSO because the requirements of strong consistency are more stringent. In strong consistency, the loads, stores, and atomic load-stores of all processors are executed by memory serially in an order that conforms to the order in which these instructions were issued by individual processors. However, a machine that implements strong consistency may deliver lower performance than an equivalent machine that implements TSO or PSO.

The general guidelines for programs are as follows: programs written for PSO will work automatically on a machine running in TSO mode or on a machine that implements strong consistency; programs written for TSO will work automatically on a machine that implements strong consistency; programs written for strong consistency may not work on a TSO or PSO machine; programs written for TSO may not work on a PSO machine.

Multithreaded programs where all threads are restricted to run on a single processor will behave the same on PSO and TSO as they would on a Strongly Consistent machine.

## **2.4 Input/Output**

The SPARC architecture assumes that input/output registers are accessed via load/store alternate instruction, normal load/store instructions, coprocessor instructions, or read/write ancillary state register instructions (RDASR, WRASR). In the case in which load/store alternate instructions are used, the I/O registers can be accessed only by the supervisor.

The contents and addresses of I/O registers are dependent on the system implementation.

## 2.5 Traps

A trap is a vectored transfer of control to the operating system through a special trap table that contains the first four instructions of each trap handler. The base address of the table is established by software in an IU state register (the trap base register, TBR). The displacement within the table is encoded in the type number of each trap. Half of the table is reserved for hardware traps, and the other half is reserved for software traps generated by trap (Ticc) instructions.

A trap causes the CWP to allocate a new register window and the hardware to write the program counters into two registers of the new window. The trap handler can access the saved PC and next program counter (nPC) and, in general, can freely use the six other local registers in the new window.

### *Trap Categories*

An exception or interrupt request can cause either a precise trap, a deferred trap, or an interrupting trap.

A precise trap is induced by a particular instruction and occurs before any program-visible state is changed by the trap-inducing instruction.

A deferred trap is also induced by a particular instruction, but, unlike a precise trap, it may occur after a program-visible state is changed by the execution of one or more instructions that follow the trap-inducing instruction. A deferred trap can occur one or more instructions after the trap-inducing instruction is executed. An implementation must provide sufficient supervisor-readable state (called a deferred-trap queue) to enable it to emulate an instruction that caused a deferred trap and to correctly resume execution of the process containing that instruction.

An interrupting trap can be due to an external interrupt request not directly related to any particular instruction, or it can be due to an exception caused by a particular previously executed instruction. An interrupting trap is neither a precise trap nor a deferred trap. An implementation need not necessarily provide sufficient state to emulate an instruction that caused an interrupting trap.

User-application programs do not “see” traps unless they install user trap handlers for those traps via calls to supervisor software. Also, the treatment of implementation-dependent machine-check exceptions can vary across systems. Therefore, SPARC lets an implementation define alternative trap models for particular exception types.

The SPARC default trap model must be present in all implementations. It states that all traps must be precise, except for:

- ☐ Floating-point or coprocessor traps, which may be deferred.
- ☐ Machine-check or non-resumable-error exceptions, which may be deferred or interrupting.
- ☐ Machine-check or non-resumable-error exceptions on the second access of a two-memory-access load/store instruction, which may be interrupting.

See Chapter 12 for information on how the SuperSPARC processor handles traps.

## 2.6 SPARC Reference MMU

The SPARC reference MMU architecture is designed for use with SPARC processors and is optional; systems may not need an MMU or may need an MMU with different characteristics. The MMU architecture enables single-chip MMU implementations to perform general-purpose memory management that efficiently supports a large number of processes running a wide variety of applications. The reference MMU uses three levels of page tables in main memory to store full translation information, and page table entries are cached in the MMU to provide quick translation.

The MMU features:

- ☐ 32-bit virtual address.
- ☐ 36-bit physical address.
- ☐ Fixed 4K-byte page size.
- ☐ Support for sparse address spaces with three-level map.
- ☐ Support for large linear mappings (4K-, 256K-, 16M-, and 4G-byte).
- ☐ Support for multiple contexts.
- ☐ Page-level protections.
- ☐ Hardware miss processing.

The reference MMU architecture specifies both the behavior of the MMU hardware and the organization and contents of the tables in the main memory required to support it.

The SuperSPARC processor contains an on-chip SPARC reference MMU. See Chapter 9.



## **SuperSPARC Introduction**

---

The SuperSPARC processor (SSP) is a highly integrated high-performance implementation of the SPARC RISC architecture. It is a single-chip processor implemented in full custom BiCMOS technology. It is intended for use in a broad spectrum of system environments: from large-scale multiprocessor systems to low-cost single-user workstations and high-performance embedded control applications.

<b>Topic</b>	<b>Page</b>
<b>3.1 High Integration .....</b>	<b>3-2</b>
<b>3.2 High-Performance Implementation Architecture .....</b>	<b>3-7</b>



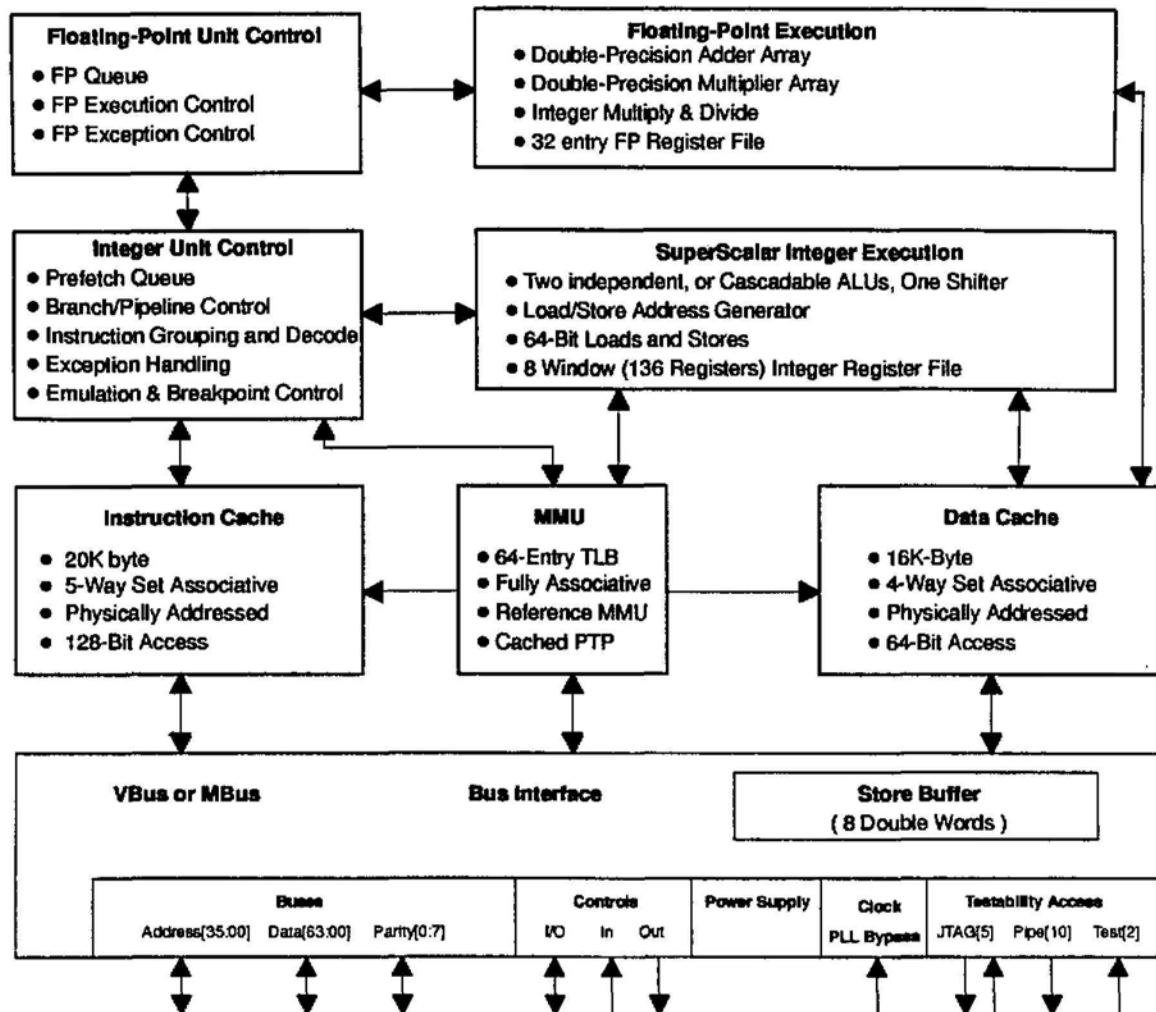
### **3.1 High Integration**

The SuperSPARC processor integrates most of the support functions normally required to build a SPARC-based system and has the following features:

- ☐ Integer Unit
- ☐ Memory Management Unit
- ☐ Floating-Point Unit
- ☐ Instruction Cache
- ☐ Data Cache
- ☐ Store Buffer
- ☐ External Cache Support
- ☐ Multi-Processor Cache Coherence Support
- ☐ Hardware Breakpoints
- ☐ JTAG Testability Access

See Figure 3-1, which is a simplified block diagram of the SuperSPARC implementation of the SPARC architecture.

Figure 3–1. Functional Block Diagram



### 3.1.1 Integer Unit

The fully SPARC-compatible on-chip integer unit has a high-performance superscalar (multiple instructions per cycle) design. Up to two operator compute instructions and one memory access instruction can be executed in each cycle. The 136 integer registers are divided into eight register windows and eight global registers. See Chapter 4 Chapter 5, and Chapter 10 contain detailed descriptions of the integer unit's operation.

### **3.1.2 Memory Management Unit**

SuperSPARC includes an implementation of the SPARC reference memory management unit (MMU). The MMU's 64-entry translation lookaside buffer (TLB) translates virtual to physical addresses. The second-level page table pointer (PTP) and the root pointer are cached to reduce the time to service TLB misses. Chapter 8 has more details.

### **3.1.3 Floating-Point Unit**

The on-chip SPARC floating-point unit (FPU) and controller provides high-performance single- and double-precision floating-point arithmetic functions and performs integer multiply and divide instructions. Chapter 11 provides a detailed view of floating-point operation.

### **3.1.4 Instruction Cache**

The large five-way set-associative instruction cache increases performance and reduces the demands on an external memory system. The cache has 20K-bytes of total storage capacity. The cache is a physically addressed cache and is non-writable but is kept consistent with the data cache and external memory through extensive cache-coherence support.

### **3.1.5 Data Cache**

An on-chip data cache ensures the single-cycle fast execution of load and store instructions that is critical to high-performance reduced instruction set computer (RISC) processors. The four-way set associative cache has 16K-bytes of total storage. This cache enforces cache coherence with other caches in a system. The data cache is a physically addressed cache and, depending on the system environment, works in either write-through or copy-back mode. The behavior of the data cache is further explained in Chapter 10.

### **3.1.6 Store Buffer**

The store buffer reduces the latency on store instructions. Its eight-entry FIFO queue holds the data until it can be written out to the external cache and/or memory. Each entry can hold the data from a single store instruction. This buffering allows the pipeline to continue execution, thereby increasing performance.

### **3.1.7 External Cache Support**

The MultiCache Controller (MXCC), SuperSPARC's optional external cache controller chip, implements a large, directly mapped, physically addressed external cache. The MXCC serves as a single-chip interface to the level-2 MBus standard or as an interface to XBus, a packet-switched bus allowing connection to a variety of system buses.

External cache data is stored in fully pipelined cache RAMs. The SuperSPARC chips support SPARC's total store ordering (TSO) and partial store ordering (PSO) memory models. See Chapter 8 and *The SPARC Architecture Manual* for further details.

### 3.1.8 Multiprocessor Cache Coherence Support

The SuperSPARC chips provide built-in multiprocessor cache coherence. The protocol supports multiple-cached copies of shared data.

Bus snooping implements the coherence algorithms. All coherence protocols are based on physical addresses.

See Chapter 16 for further details.

### 3.1.9 Hardware Breakpoints

On-chip hardware breakpoints with code and data access simplify software debugging and reduce system-development time. A single code or data access breakpoint can be set on a virtual or physical address.

A 16-bit instruction counter and a 16-bit cycle counter debug and analyze performance. These counters can be used to generate breakpoints.

When these breakpoints occur, they all have programmable actions. They can generate exceptions or interrupts or toggle an external pin to help trigger external analysis equipment.

See Chapters 17, 18, and 19 for further details.

### 3.1.10 JTAG Emulation

Through the SuperSPARC processor's *JTAG IEEE P1149.1* asynchronous scan interface, the state of the processor can be viewed or modified without changing other processor states. The processor can be single-stepped through a program, and all processor states can be observed after each instruction group. This interface can also be used to view or modify registers or system memory.

Only a JTAG control device with appropriate software is required to use this facility; no test pod or other specialized hardware is required.

See Chapter 15 for further details.

### **3.1.11 Full Testability**

SuperSPARC is designed to be a highly testable device. JTAG scan gives access to internal data paths and control logic for testing. Large internal arrays are not in the scan chain but can still be tested through the serial JTAG interface. An automatic power-up self-test can be initiated with or without any external scan hardware. This, along with functional testing of the arrays, assures that the device is operating. Boundary scan can be used to perform board-level interconnect testing.

## 3.2 High-Performance Implementation Architecture

In order to push beyond the improvement from clock rate, the infrared architecture has been optimized to execute multiple instructions simultaneously and critical instructions quickly. These architectural features increase the average number of instructions executed per cycle by a factor of two for integer programs. A greater improvement may be found for floating-point-bound programs.

SuperSPARC typically executes programs from its cache at about 1.4 to 1.6 instructions per cycle (IPC), or about 0.7 to 0.6 clocks per instruction (CPI). This figure decreases to about 1.1 IPC for large programs not fully contained in the cache. Floating-point performance is generally higher.

The major implementation architecture optimizations are outlined below.

### 3.2.1 Multiple-Instructions-per-Cycle Execution

SuperSPARC can issue up to three instructions simultaneously. Certain rules determine how many of the available instructions can be executed in any particular cycle. These rules are fully described in Chapter 5.

### 3.2.2 Fast Load and Store Instructions

All load and store instructions operate in a single clock cycle when the referenced data is present in the on-chip data cache. This includes 64-bit transfers and floating-point transfers. When the data is not present in the on-chip data cache, a five-cycle penalty is imposed to access the external cache. Each cache miss reads a block (32 bytes) of data from the external cache. These bus transactions are fully pipelined. The processor can use this data as soon as it arrives from the bus.

When using external cache memory, no miss penalty is incurred for normal store misses. An internal store buffer holds the store transaction and allows the pipeline to continue.

The instruction immediately following a load may use the data without incurring any delay. There are some cases of interlocks between the load instruction and the following address calculations (described in Section 5.4). The external bus is pipelined for high performance. It is capable of delivering data from different addresses on successive cycles.

Because all of SuperSPARC's caches are physically addressed and are fully coherent, there is no need to flush cached entries, and virtual address aliasing conditions do not exist. Eliminating flushing overhead can boost performance significantly.

### 3.2.3 Floating-Point Implementation

The SuperSPARC FPU allows one floating-point operation and one memory reference to be issued in every clock cycle. SuperSPARC supports single- and double-precision operations but not extended or quad-precision. The FPU maintains a four-entry deferred trap queue (FQ) from which FPopS are executed. Some operations require more execution cycles than others; for example, FDIV (floating-point divide) and FSQRT (floating-point square root) use many more cycles than FADD. SuperSPARC FPU also handles the integer multiply and divide operations. Floating-point instructions are executed in the order in which they are issued by the processor, allowing no out-of-order completion. Register dependencies can delay the execution stream, and exceptions can interrupt the pipeline, sometimes requiring instruction aborts. SuperSPARC handles all cases of normalization and register alignments for double-precision arithmetic, directly in hardware. SuperSPARC does not generate unfinished exceptions (unfinished\_FPop trap).

Chapter 11 provides details about the FPU.

## **Register Summary**

---

This chapter details most of the SuperSPARC registers.

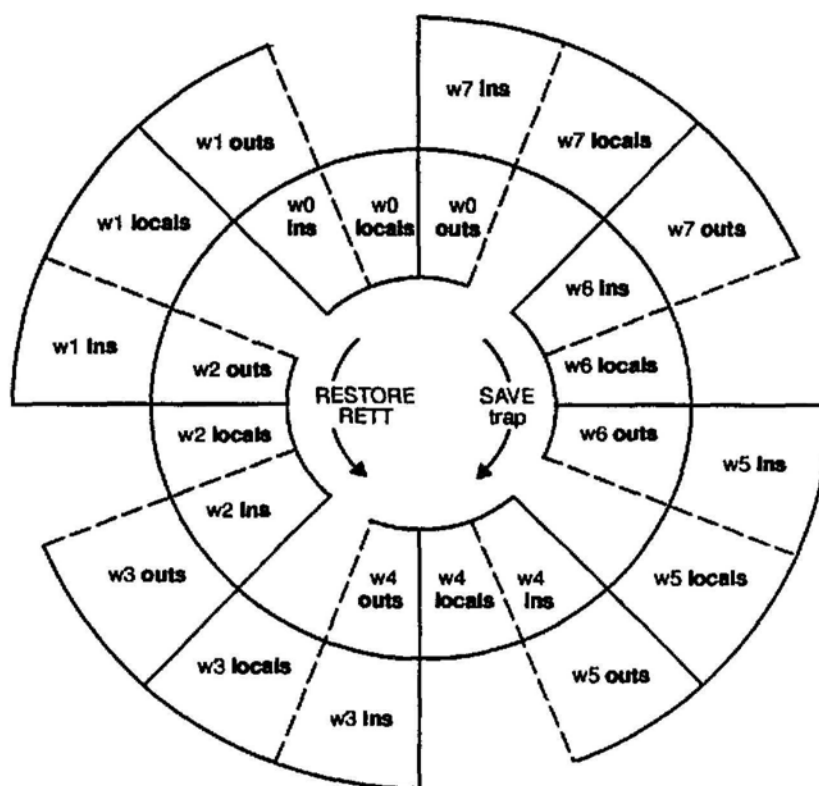
<b>Topic</b>	<b>Page</b>
<b>4.1 Integer Unit <i>r</i> Registers</b> .....	<b>4-2</b>
<b>4.2 Processor State Register</b> .....	<b>4-4</b>
<b>4.3 Window Invalid Mask</b> .....	<b>4-6</b>
<b>4.4 Trap Base Register</b> .....	<b>4-7</b>
<b>4.5 Multiply/Divide Register (Y)</b> .....	<b>4-8</b>
<b>4.6 Program Counters</b> .....	<b>4-9</b>
<b>4.7 Ancillary State Registers</b> .....	<b>4-10</b>
<b>4.8 Floating-Point <i>f</i> Registers</b> .....	<b>4-11</b>
<b>4.9 Floating-Point State Register</b> .....	<b>4-12</b>
<b>4.10 Floating-Point Queue</b> .....	<b>4-16</b>



## 4.1 Integer Unit *r* Registers

The SuperSPARC processor (SSP) provides eight register windows in its integer unit. The *r* registers are divided into 128 window registers and eight global registers. Each register is 32 bits wide. The window registers are divided into eight sets of 16 registers. At any time an instruction can access the eight global registers and a 24-register window of *r* registers. A register window comprises eight *in* registers and eight local registers of a particular register set, along with the eight *in* registers of the adjacent register set—addressable as the window's *out* registers. See Figure 4–1.

Figure 4–1. Windowed *r* Registers



Instructions that access double-words in *r* registers require an aligned pair of registers. The least significant bit of an *r* register index in these instructions is reserved and should be set to zero. An attempt to use a double-word load or store to a misaligned (odd) destination register will cause an *illegal\_instruction* trap.

The current window into the *r* registers is given by the current window pointer (CWP) in the processor state register. The window indexing is as shown in Table 4-1.

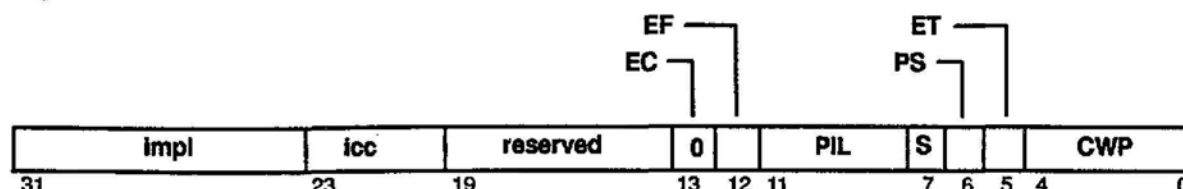
*Table 4-1. Window Addressing*

Windowed Register Index	<i>r</i> Register Index
in[0] – in[7]	r[24] – r[31]
local[0] – local[7]	r[16] – r[23]
out[0] – out[7]	r[8] – r[15]
global[0] – global[7]	r[0] – r[7]

## 4.2 Processor State Register

The 32-bit processor state register (PSR) holds key status and control information. The instructions that modify the PSR's fields include SAVE, RESTORE, Ticc, RETT, and any instructions that modify the condition codes. The privileged instructions RDPSR and WRPSR read and write the PSR directly. See Figure 4-2.

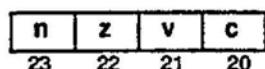
Figure 4-2. Processor State Register



The PSR contains the following fields:

- Impl** Implementation. Bits 24 through 31 contain SuperSPARC's version number. SuperSPARC's implementation number is 0x40. The WRPSR instruction does not affect the contents of this field.
- icc** Integer Condition Codes. Bits 20 through 23 contain SuperSPARC's condition codes. These bits are modified by WRPSR and instructions ending in cc. Conditional branch and trap instructions base their control transfer on these bits, which are defined as shown in Figure 4-3.

Figure 4-3. Integer Condition Codes (icc)



- icc.n** Negative. Bit 23 indicates whether the 32-bit 2's complement arithmetic logic unit (ALU) result was negative for the last instruction that modified the icc field. 1 = negative, 0 = not negative.
- icc.z** Zero. Bit 22 indicates whether the 32-bit ALU result was zero for the last icc-modifying instruction. 1 = zero, 0 = nonzero.
- icc.v** Overflow. Bit 21, the overflow bit, indicates whether the ALU result was representable in 32-bit 2's complement notation for the last icc-modifying instruction. The overflow bit is also set if a tagged operation is performed on non-tagged operands. 1 = overflow, 0 = no overflow.

<b>icc.c</b>	Carry. Bit 20 indicates whether a 2's complement carry (borrow) out of bit 31 resulted from the last icc-modifying addition (subtraction). 1 = carry, 0 = no carry.
<b>reserved</b>	Reserved. Bits 14 through 19 are reserved. When read by a RDPSR, these bits return zero. A WRPSR should write only 0's to this field.
<b>EC</b>	Enable Coprocessor. Bit 13 determines whether a coprocessor is enabled. In SuperSPARC, this bit is permanently set to zero. Coprocessor instructions will cause a cp-disabled trap. If a WRPSR instruction attempts to set the EC bit, an illegal_instruction trap will be generated.
<b>EF</b>	Enable FPU. Bit 12 determines whether the FPU is enabled. When disabled, a floating-point instruction will cause a fp-disabled trap. 1 = enabled, 0 = disabled.

**Note:**

Software can use the EF bit to determine whether the floating-point unit (FPU) is used by a particular process. If the FPU is unused by a process, the *f* registers need not be saved across a context switch.

<b>PIL</b>	Processor Interrupt Level. Bits 8 (LSB) through 11 (MSB) determine the external interrupt priority level above which SuperSPARC will accept external interrupts.
<b>S</b>	Supervisor. Bit 7 determines whether the processor is in supervisor or user mode. 1 = supervisor mode, 0 = user mode.
<b>PS</b>	Previous Supervisor. Bit 6 contains the value of the S bit at the time of the most recent trap.
<b>ET</b>	Enable Traps. Bit 5 determines whether traps are enabled. A trap will automatically set ET to zero, disabling further traps. While traps are disabled (ET = 0), interrupt requests are ignored, and an exception trap causes SuperSPARC to halt execution enter error mode and take a watchdog reset. 1 = traps enabled, 0 = traps disabled.
<b>CWP</b>	Current Window Pointer. Bits 0 (LSB) through 4 (MSB) contain the CWP. This is a pointer to the current active register window. The hardware increments the CWP on a RESTORE and RETT and decrements it on SAVE and trap.

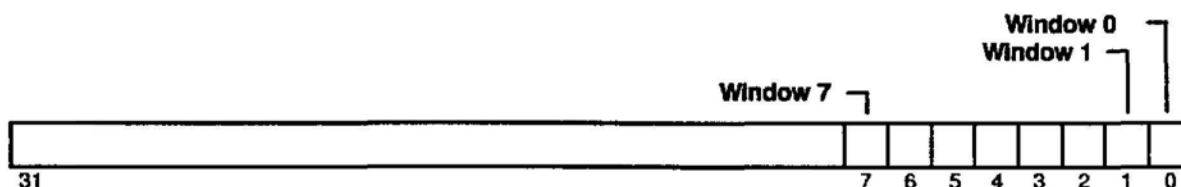
### 4.3 Window Invalid Mask

The Window Invalid Mask (WIM) register is controlled by supervisor software. Hardware then uses the WIM to determine which window(s) cause window overflow and underflow traps on SAVE, RESTORE, or RETT instructions.

Each bit in the WIM register corresponds to a register window. WIM[n] corresponds to the register set addressed when CWP = n. If WIM[n] = 1, window n is marked invalid. Should the CWP be decremented into an invalid window by a SAVE instruction, a window\_overflow trap is generated. Should a RESTORE or RETT instruction increment the CWP into an invalid window, a window\_underflow trap is generated.

The WIM is read and written by the privileged instructions RDWIM and WRWIM, respectively. Bits 31–38, which correspond to unimplemented windows, are read as zeros and are unaffected by writes. See Figure 4–4.

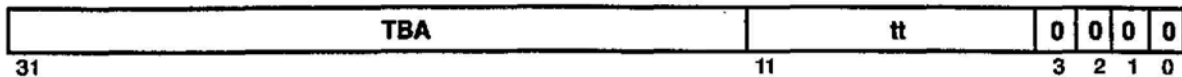
Figure 4–4. Window Invalid Mask



## 4.4 Trap Base Register

When a trap occurs, the program counter (PC) is loaded with the contents of the trap base register (TBR). (See Figure 4-5.) The TBR is a pointer into the trap table that contains the trap-handler address. The privileged instruction RDTBR reads the entire register. Bits 0 through 3 are zeros, and the WRTBR instruction should always be issued with zeros in this field. The TBR contains the following fields shown in Figure 4-5.

Figure 4-5. Trap Base Register



**TBA** Trap Base Address. Bits 12 through 31 contain the 20-bit trap base address. This field is written by the privileged WRTBR instruction. The trap base address is usually established by supervisor software.

**tt** Trap Type. Bits 4 through 11 contain the eight-bit trap type field. This eight-bit field is written by the hardware, based on the type of trap taken, and provides an offset into the trap table. The tt field retains its value until the next trap and is not affected by the WRPSR instruction. The tt field can be directly manipulated by Ticc instructions.

## **4.5 Multiply/Divide Register (Y)**

The 32-bit *Y* register contains the most significant word of the double-precision product of an integer multiply using an SMUL, SMULcc, UMUL, UMULcc, or MULScc instruction. The *Y* register also holds the most significant word of the double-precision dividend of an integer divide using a SDIV, SDIVcc, UDIV, UDIVcc instruction. The *Y* register is read and written by the non-privileged RDY and WRY instructions, respectively.

#### **4.6 Program Counters (PC, nPC)**

The 32-bit PC contains the address of the instruction currently being executed in SuperSPARC's integer unit. The next program counter (nPC) holds the address of the next instruction to be executed (assuming a trap does not occur).

The nPC facilitates delayed control transfers. The delay instruction is executed (unless the control transfer annuls it) before control transfers to the target. During execution of the delay instruction, the nPC points to the target of the control transfer instruction.

Both the PC and nPC are available in the local registers of the trap handler after a trap. This allows the handler to choose between resuming the program execution from the trapping instruction or from the instruction following the trap-causing instruction.



## **4.7 Ancillary State Registers**

SuperSPARC has no ancillary state registers, but encodes the STBAR and SIGM instructions as special cases of RDASR, the instruction for reading ancillary state registers. (See Chapter 7.) The STBAR instruction is implemented in the RDASR-reserved instruction space and is equivalent to reading ancillary state register (ASR) 0x0f (RDASR 0x0f, %g0). The SIGM instruction is implemented in the RDASR implementation-dependent extended opcode space as defined by Version 8 of the SPARC architecture. SIGM is equivalent to reading ASR 0x1f (RDASR 0x1f, %g0).

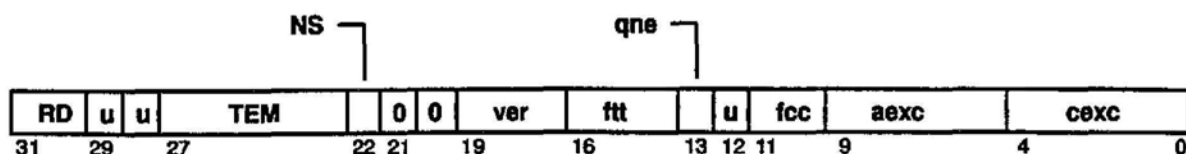
## **4.8 Floating-Point $f$ Registers**

SuperSPARC has 32 32-bit floating point  $f$  registers, numbered from  $f[0]$  through  $f[31]$ . There is no windowing of the  $f$  registers; floating-point instructions have access to all 32 registers at all times. A single  $f$  register can hold one single-precision operand. A double-precision operand requires an aligned pair of  $f$  registers. Thus the  $f$  registers can hold a maximum of 32 single-precision or 16 double-precision operands. Instructions that access a floating-point double in the  $f$  registers assume double alignment, and the least-significant bit of a double-precision  $f$  register specifier is reserved and should be set to zero.

## 4.9 Floating-Point State Register

The 32-bit floating-point state register (FSR) fields contain mode and status information. The FSR is read and written by the STFSR and LDFSR, respectively. See Figure 4-6.

Figure 4-6. Floating-Point State Register (FSR)



The FSR contains the following fields:

**RD** Rounding Direction. Bits 30 and 31 selects an ANSI/IEEE Standard 754-1985 rounding direction for floating-point results. See Table 4-2.

Table 4-2. Rounding Direction (RD) Field of FSR

RD	Round Toward:
00	Nearest (even, if tie)
01	0
10	$+\infty$
11	$-\infty$

**u** Unused. Bits 29, 28, and 12 are unused in the SuperSPARC implementation. To ensure future compatibility, software should always issue a LDFSR with zeros in these bits.

**TEM** Trap Enable Mask. Bits 23 through 27 represent the FPU's trap enable mask. Each of the five bits represents one of the five floating-point exceptions that can be indicated in the current exception (cexc) field. If a floating-point operate (FPop) instruction generates one or more exceptions, and the TEM bit corresponding to one or more of the exceptions is 1, an fp\_exception trap is generated. A TEM value of zero prevents that exception type from generating a trap. See Figure 4-7.

Figure 4-7. Trap Enable Mask (TEM) Field of the FSR

NVM	OFM	UFM	DZM	NXM
27	26	25	24	23

**TEM.NVM** Invalid Trap Mask. Bit 27 represents the invalid operation trap mask. An invalid operation exception will occur when an improper operand is supplied to a FPop instruction. For example  $0 \div 0$  is invalid. 0 = disable invalid operation trap, 1 = enable invalid operation trap.

**TEM.OFM** Overflow Trap Mask. Bit 26 represents the overflow trap mask. An overflow exception will occur when the rounded result would be larger than the largest normalized number in the result format. 0 = disable overflow trap, 1 = enable overflow trap.

**TEM.UFM** Underflow Trap Mask. Bit 25 represents the underflow trap mask. An underflow exception will occur when the rounded result is inexact and would be smaller than the smallest normalized number in the result format. 0 = disable underflow trap, 1 = enable underflow trap.

**TEM.DZM** Divide By Zero Trap Mask. Bit 24 represents the divide by zero trap mask. A divide by zero exception will occur for all  $X \div 0$ , where X is normalized or subnormal but not zero ( $0 \div 0$  will not generate a divide-by-zero exception). 0 = disable divide by zero trap, 1 = enable divide by zero trap.

**TEM.NXM** Inexact Trap Mask. Bit 23 represents the inexact trap mask. An inexact exception will occur if the rounded result of a FPop instruction differs from the infinitely exact result. 0 = disable inexact trap, 1 = enable inexact trap.

**NS** Non-Standard Mode. Bit 22 represents non-standard mode execution. SuperSPARC ignores the NS bit; it can be read or written but has no effect on floating-point execution. SuperSPARC adheres to ANSI/IEEE Standard 754-1985 and is unaffected by the NS bit's setting.

**ver** Version. Bits 17 through 19 represent the FPU version. On SuperSPARC, this field is always zero.

**ftt** Floating-Point Trap Type. Bits 14 through 16 identify the floating-point exception trap type. The ftt field encodes the type of exception that occurred until a STFSR or another FPop is executed. The ftt can be read by the STFSR instruction, but the LDFSR instruction does not affect the field. The exception types are encoded as shown in Table 4-3.

Table 4-3. Floating-Point Trap Type (ftt) Field of FSR

ftt	Trap Type
000	None
001	IEEE_754_exception
010	unfinished_FPop
011	unimplemented_FPop
100	sequence_error
101	hardware_error
110	invalid_fp_register
111	reserved

**qne** Queue Not Empty. Bit 13 indicates whether SuperSPARC's four-entry floating-point queue is empty after a floating-point exception or STDFQ instruction has been issued. The qne bit can be read by the STFSR instruction but is unaffected by the LDFSR instruction. 0 = queue is empty, 1 = queue is not empty.

**fcc** Floating-Point Condition Codes. Bits 10 and 11 contain the floating-point condition codes. (See Table 4-4.) These bits are modified by the FCMP and FCMPE floating-point compare instructions. The fcc field can be read and written by the STFSR and LDFSR instructions, respectively. Floating-point branches (FBfcc) base their control transfer on this field. In the following table, the question mark (?) denotes an unordered relation, which is true if either  $f_{rs1}$  or  $f_{rs2}$  is signaling NaN.

Table 4-4. Floating-Point Condition Codes (fcc) Field of FSR

fcc	Relation
00	$f_{rs1} = f_{rs2}$
01	$f_{rs1} < f_{rs2}$
10	$f_{rs1} > f_{rs2}$
11	$f_{rs1} ? f_{rs2}$ (unordered)

**aexc**      **Accrued Exceptions.** Bits 5 through 9 represent the accrued exception field of the floating-point unit. (See Figure 4–8.) After an FPop completes, the TEM and the *cexc* fields are logically ANDed together. Should the result be nonzero, an *fp\_exception* trap is generated, or else the new *cexc* is logically ORed into the *aexc* field. While traps are masked by the TEM field, exceptions are accumulated in the *aexc* field. The bits in the *aexc* field assume the same definition as those in the TEM field.

Figure 4–8. Accrued Exception Bits (*aexc*) Field of the FSR

nva	ofa	ufa	dza	nxa
9	8	7	6	5

**cexc**      **Current Exceptions.** Bits 0 through 4 indicate that one or more IEEE 754 floating-point exceptions were generated by the most recently executed floating-point instruction. (See Figure 4–9.) The *cexc* field is automatically cleared by the execution of the next floating-point instruction. The bits in the *cexc* field assume the same definition as those in the TEM field.

Figure 4–9. Current Exception Bits (*cexc*) Field of the FSR

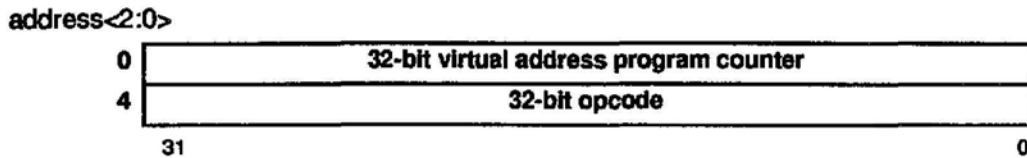
nvc	ofc	ufc	dzc	nxc
4	3	2	1	0

## 4.10 Floating-Point Queue

SuperSPARC's four-entry-deep floating-point queue contains enough information to implement resumable, deferred floating-point traps. All FPops are written to the queue (FPops do not include memory references, FSR operations, or the privileged STDFQ). The contents of the floating-point queue should only be stored in exception mode. A STDFQ when not in exception mode will cause SuperSPARC to hold the pipeline until all floating-point operations have completed and subsequently store information regarding the last completed floating-point operation.

An STDFQ will return a double containing the virtual address and 32-bit opcode of the instruction at the front of the queue. The format is as shown in Figure 4-10.

Figure 4-10. Floating-Point Queue Format



## **Principles Of Operation**

---

Efficient use of the SuperSPARC processor's (SSP's) pipeline is a major factor in system performance. This chapter describes SuperSPARC's pipeline operation and how to use it efficiently. Pipeline fundamentals are introduced, and the intricacies of SuperSPARC's pipeline are illustrated through diagrams and examples. Code generation strategies are also discussed, and guidelines are given to obtain maximum performance using SuperSPARC's superscalar pipeline.

<b>Topic</b>	<b>Page</b>
<b>5.1 Introduction .....</b>	<b>5-2</b>
<b>5.2 Pipeline Fundamentals .....</b>	<b>5-4</b>
<b>5.3 Pipeline Example Overview .....</b>	<b>5-7</b>
<b>5.4 Memory References .....</b>	<b>5-8</b>
<b>5.5 Floating-Point Pipeline .....</b>	<b>5-13</b>
<b>5.6 Conditional Branches in the Pipeline .....</b>	<b>5-18</b>
<b>5.7 Procedure Call and Return .....</b>	<b>5-23</b>
<b>5.8 Exceptions and the Pipeline .....</b>	<b>5-24</b>



## 5.1 Introduction

This chapter will introduce instruction execution in an abstract framework. Understanding the subtle details of the SuperSPARC processor's instruction execution aids in realizing the processor's full potential for high performance.

A program's instructions are arranged in the order in which they will be visited by the program counter (PC). SPARC program order is defined by the SPARC Architecture. Most processor implementations, including SuperSPARC, do not actually perform all the steps of a program in program order, but must give every appearance of having done so. Two implementation techniques used in the SuperSPARC processor, pipelined execution and superscalar execution, deviate from program order. In addition, SuperSPARC's store buffer can delay stores relative to loads from other addresses, and some memory systems may perform operations out of order.

Since the programmer relies on program order when constructing the program, the processor must maintain the illusion of program order at all times. This is most difficult around traps.

A program is executed for its effects, for the changes it makes to registers and memory, and for the I/O it performs. One technique used in SuperSPARC to exhibit program order is to speculatively execute instructions but cancel their effects before registers, memory, or I/O are affected in case the instructions are not needed. A cancelled instruction is called a *squashed*, or *aborted*, instruction.

The execution of an instruction can be broken down into several steps, including fetching, decoding, computing, and storing the results. Rather than completing all these steps for one instruction before starting the next, the steps for different instructions can be overlapped. The first step of each new instruction can be performed while the previous instruction is on the second step and the second previous instruction is on the third step. In this manner, several instructions are being executed simultaneously, thereby reducing the average execution time per instruction. This technique is called *pipelining*.

A pipelined instruction sometimes requires an input that is being computed by one of the several previous instructions that have begun execution but are not yet complete. In this case, the instruction might wait until all the required inputs have been stored as results before beginning execution. A faster scheme is to issue an instruction as soon as it is certain all of its inputs will be available in time to be used. This is called *data forwarding*, or *forwarding*. For this to work, the processor must contain data paths to route the results of earlier instructions directly to the inputs of computations for later instructions.

Pipelining reduces the time required to execute a sequence of instructions, because instructions can be started more frequently. Computation-per-unit time is measured as *throughput*. Pipelining improves throughput. However, pipelining does not improve the amount of time required to execute a single instruction. This time to produce an answer is called *latency*.

A *superscalar* microprocessor can issue and execute two or more instructions in parallel. A superscalar design increases the amount of work a processor can perform per cycle, thereby increasing its efficiency. Ideally, with  $N$  concurrent, simultaneous operations, the performance of a superscalar processor should exceed that of a scalar processor by a factor of  $N$ . Data dependencies, procedural dependencies, and resource conflicts, however, limit the magnitude of this performance boost. To reduce these effects, SuperSPARC has logic to dynamically schedule instructions and duplicate functional units. These features maximize SuperSPARC's capability of issuing three instructions per cycle. As a result, SuperSPARC's performance typically exceeds that of a scalar processor by 40%.

SuperSPARC is a single-pipeline, three-way, dynamic superscalar microprocessor. It can issue up to three instructions in each clock cycle (three-way superscalar). It decides how many instructions to run by examining the next few instructions available, rather than by issuing groups of instructions in memory that had been previously marked by the compiler or the assembly language programmer. This is called dynamic grouping. Finally, the group of instructions chosen are executed together in lock-step; if one instruction of the group is delayed (for example, for a cache miss), the entire group waits. This is as if there were a single pipeline three instructions wide. This technique greatly simplifies trapping and error recovery.

## 5.2 Pipeline Fundamentals

The SuperSPARC processor's pipeline consists of eight stages, which execute in four clock cycles. Each stage has unique functions that contribute to completing instructions in the group. Different types of instructions are supported by different functions in several of the pipeline stages. The SuperSPARC pipeline stages are:

F0 F1 D0 D1 D2 E0 E1 WB
-------------------------

Each stage is described in the following sections.

### 5.2.1 F0/F1 (Fetch)

All instructions must be fetched before they are executed. However, not every instruction is fetched in the cycle immediately preceding its execution. An instruction may be prefetched and placed in the instruction queue. The fetch stages (F0/F1) of the pipeline manage the instruction queue, including fetching and prefetching required instructions from memory. Not every fetched instruction is executed. Some instructions may be discarded if a control transfer instruction (branch) changes the flow of execution. Up to 128 bits (four instructions) may be read from the instruction cache in every cycle. These instructions enter into the instruction queue and can be removed at a maximum rate of three instructions per cycle.

### 5.2.2 D0 (Grouping)

The D0 stage selects the first one, two, or three instructions from the instruction queue to form a group. This selection depends on the set of instruction candidates available at the head of the instruction queue prefetch buffer, as well as the current state of the processor pipeline. The grouping rules used to form this selection are described in Section 6.6. These instructions must be taken in order from the queue. SuperSPARC does not execute instructions out of order.

Once a group of instructions is selected, D0 identifies the single memory reference instruction in the group (if there is one) and latches the corresponding register index. D0 forms extension words based on the immediate values for memory reference and control transfer instructions' displacements. D0 identifies cascade conditions (integer instruction data dependencies within and between instruction groups) and inserts pipeline bubbles when necessary. A bubble is a cycle where no instruction is executed. This cycle is necessary when the required data is not available.

### 5.2.3 D1 (Resource Allocation)

D1 assigns available resources within the integer unit to individual instructions in the group selected during D0. All cases of data forwarding (or bypass) are resolved in this stage. All operand register indexes are selected and assigned to individual register file ports. These resources remain constant throughout the execution of the instructions.

The two address registers selected during D0 are read via two dedicated register file ports during D1. This data is used in D2 to compute a load or store virtual address. The data for these may also be forwarded from currently executing instruction groups.

Branch target addresses are generated in D1, taken from the extension words selected in D0 and the PC value of the branch instruction within the group. Next PC (nPC) values are also generated.

### 5.2.4 D2 (Read Operands)

Stage D2's primary function is to read the operand registers selected in the preceding D1 stage. In addition, the address operands read during D1 will be combined in the virtual address adder. The result is a 32-bit virtual address that will be used to reference the Memory Management Unit (MMU) and data cache in subsequent stages. During D2, any data forwarding paths required for execution will be set up to transfer data in cycles that follow.

### 5.2.5 E0 (Execute First Stage)

The SSP has two execution stages. E0 is the primary execution stage. Most arithmetic logic unit operations (ALUops) complete in E0. During E0, the data operands read from the register file during D2 are passed through one of two ALUs or the shifter. A maximum of two integer results can be generated in E0. Only one may be generated by the shifter. These results are then presented as input to the E1 cascaded ALU and sent into many forwarding paths.

For memory references, the virtual address generated in D2 is used in E0 to begin accessing the translation lookaside buffer (TLB) and the data cache. Only the low-order 12 bits of the virtual address are needed to begin cache lookup. The high-order bits are supplied by the MMU in the E1 phase for tag comparison with the physically cached data. The MMU must inform the data cache unit in E0 if there is an access exception in the current group of instructions. The integer unit (IU) is also informed in E1 stage about the access exception. If it is not yet known whether an exception must be reported to the current group (due to TLB or cache misses), the pipeline is stalled at this stage until all exception sources have been resolved.

Floating-point operations are dispatched to the floating-point unit (FPU) during E0. From this point forward, floating-point operations (FPops) execute in the FP pipeline explained in Section 5.5.

### 5.2.6 E1 (Execute Second Stage)

The second stage of execution can generate at most one additional integer ALU result, made up of one computed result from E0, plus one “pass-through” value, read or bypassed by D2 (no “double cascades” are allowed). This result is generated in the cascaded ALU. The computed results from the E0 ALU or shifter are used as inputs to this ALU. All execution results except FPops from the current instruction group are available by the end of the E1 stage, including data returned from the data cache. Results generated in E1 are delayed a cycle before they can be used as address operands. Address dependencies for a load from memory result in one cycle of pipeline bubble. Condition codes generated in E1 are delayed a cycle before they can be used in resolving conditional branches.

### 5.2.7 Write Back (WB) Results

When stage E1 has completed, all non-FPop results are guaranteed to be available. The primary action in the WB pipeline stage is to write back these results into the register file. Only instructions that complete correctly with no prior exceptions are written back. The WB stage executes at the same time as the E0 stage of the next instruction group. Forwarding paths are used to transmit data between successive groups. The integer unit updates the register file during WB, and the data cache normally updates its contents when an ST instruction has appeared in E0-E1.

SuperSPARC can operate either directly connected to MBus or with the SuperSPARC MultiCache Controller (MXCC). This choice has a major impact on the behavior of store instructions. When connected to the MXCC, SuperSPARC assumes the existence of an external cache, and the SuperSPARC data cache behaves as a write-through cache, which means that all store instructions that modify the internal cache also write their data through to the external cache.

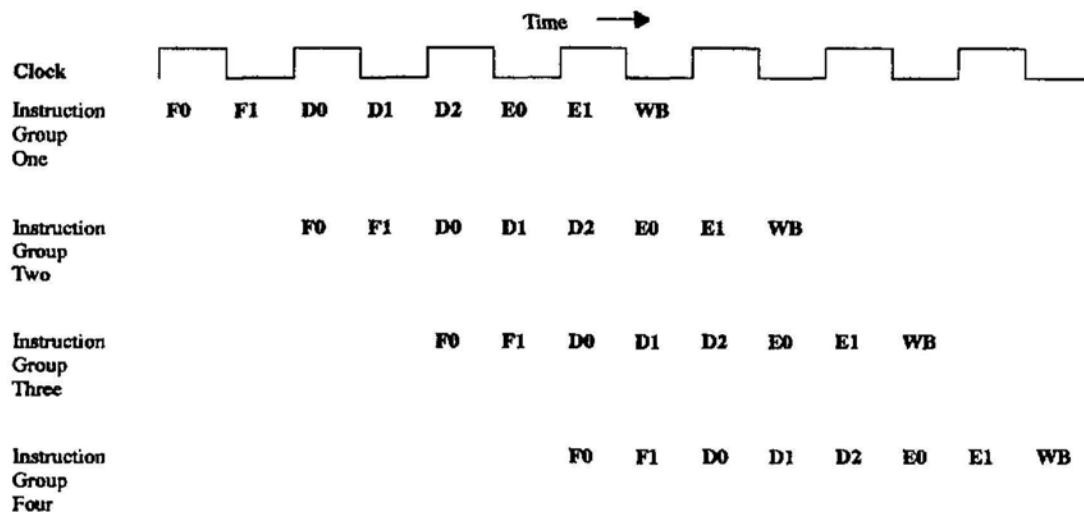
When connected directly to MBus, SuperSPARC’s data cache operates as a copy-back cache. Store data will remain in the cache until either the line containing the data is replaced or a snoop on the bus forces a copy-back. The actions taken on a snoop hit are further explained in Chapter 17. The cache also implements a write-allocate policy. Should a store miss in the cache, the block containing that data is brought in from memory. The store is then performed locally, and, consistent with the copy-back policy, memory is not updated. In this configuration SuperSPARC does not assume the presence of an external cache.

### 5.3 Pipeline Example Overview

The SuperSPARC processor's pipeline is straightforward for simple instruction sequences, for example, ALUop. The complexity increases quickly for memory reference and control transfer instructions. The following sections describe these cases in detail. Standard load and store sequences are presented first in Section 5.4, followed by floating-point operations (FPops) in Section 5.5. SAVE, RESTORE, and all forms of control transfers are then described in Section 5.7 and Section 5.6. Section 5.8 describes how the pipeline deals with exceptions.

Figure 5-1 describes how the pipeline works in simple cases. There are no pipeline stalls or bubbles that can be caused in a variety of ways. These will be dealt with later in this chapter. Figure 5-1 is similar to the pipeline diagrams used throughout the chapter to describe the operations of the processor.

Figure 5-1. Basic Pipeline Description



All pipeline stages are identified. In general, the contents of the instruction group will be indicated in the left-side heading. Significant operations and interactions are included in the boxes for individual stages.

Notice how the groups overlap in time. While executing E0 of instruction group two, WB of instruction group one and D1 of instruction group three are executing.

## 5.4 Memory References

Load and store instructions are frequent operations in SPARC programs. In a typical program, as many as 30% of the instructions are loads or stores. Since SuperSPARC executes up to three instructions per cycle, it may be required to execute a memory reference nearly every cycle. Such a task stretches the SuperSPARC processor design a great deal.

To maximize performance, SuperSPARC has removed restrictions associated with prior RISC designs. In particular, many sources of interlocks on load instructions have been removed. This allows SuperSPARC to execute a Load instruction, immediately followed by a dependent ALUop (with a register dependency on the load) in the next instruction group.

All LD and ST instructions that hit in the internal data cache execute in a single cycle. This includes all byte, half-word, word, and double-word references. Up to two other instructions may be included in the instruction along group with the memory reference. Stores are generally buffered. When SuperSPARC is used with MXCC, stores take a single cycle to execute, regardless of whether the stores hit in the cache.

### 5.4.1 Load Operation

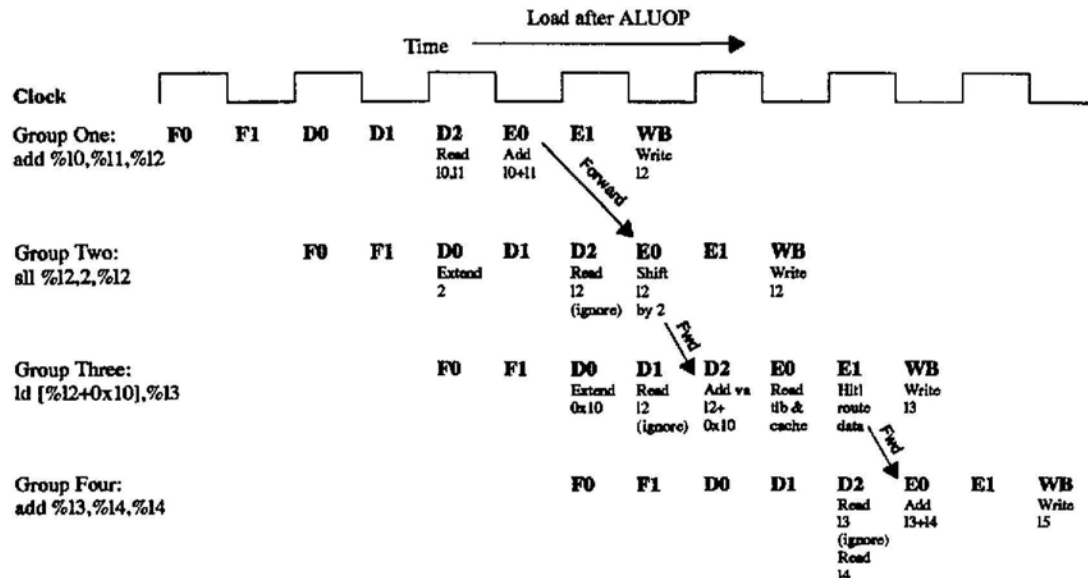
Example 5-1 shows an LD instruction surrounded by arithmetic operations. For simplicity, the sequence uses single-instruction groups, forced by the dependencies in the code. The code sequence being executed demonstrates the use of many data forwarding paths.

#### *Example 5-1. Simple Load Operation*

```
add    %10,%11,%12
!---Split (can't cascade into shifter)
sll    %12,2,%12
!---Split (address dependency)
ld     [%12+0x10],%13
!---Split (Load data dependency)
add    %13,%14,%15
```

The execution of this code sequence through the pipeline is shown in Figure 5-2.

Figure 5-2. Basic Load Pipeline Sequence (Example 5-1)



The add and shift instructions execute in E0 of group one and group two, respectively. The instructions pass data through forwarding paths from the add result into the shifter, and then from the shifter result in the virtual address adder for the load. The 0x10 offset is extended into a 32-bit value in the D1 stage. The offset extension word and the forwarded version of register %12 are added, and the result is passed to both the data cache and the MMU, which are accessed in parallel. When a hit is identified in the TLB, the physical page number is extracted and passed on to the data cache. In the meantime, the cache has completed reading all data and tags for the four lines in the set of the memory location (four-way set associative cache). The tags are compared with the physical page number from the MMU. When the proper line is identified, the appropriate bytes become the result of the load instruction. The load result is also forwarded into the next E0 execute stage for the last add instruction, and it is written into the register file in the WB stage.

If a cache or MMU miss had occurred, pipeline bubbles would have been inserted. The E0 and E1 stages of the load would have been repeated until all the misses were satisfied. If any errors had occurred, they would have been reported to the E0 stage (while it was being repeatedly executed).

Note the use of a dependent shift instruction to force a split between the add and shift. If the shift were replaced by an add, it would be considered a cascaded instruction, and the two would be grouped together. This would result in a pipeline bubble between the second add and the load, as shown in Example 5-2. The total execution time would be identical.



*Example 5-2. Load Operation With Bubble*

```

add    %10,%11,%12
add    %12,2,%12 !Cascade
!---Split (address dependency)
!bubble
!---Split (address dependency)
ld     [%12+0x10],%13
!---Split (Load data dependency)
add    %13,%14,%15

```

**5.4.2 Store Operation**

Store operations are similar to loads in many ways. The address computation, cache lookup, and MMU access are performed in the same manner as loads. The primary difference is the handling of the data. Example 5-3 demonstrates a store operation. To illustrate multiple instruction handling, the instruction sequence is more complex than the previous example.

*Example 5-3. Store Operation*

```

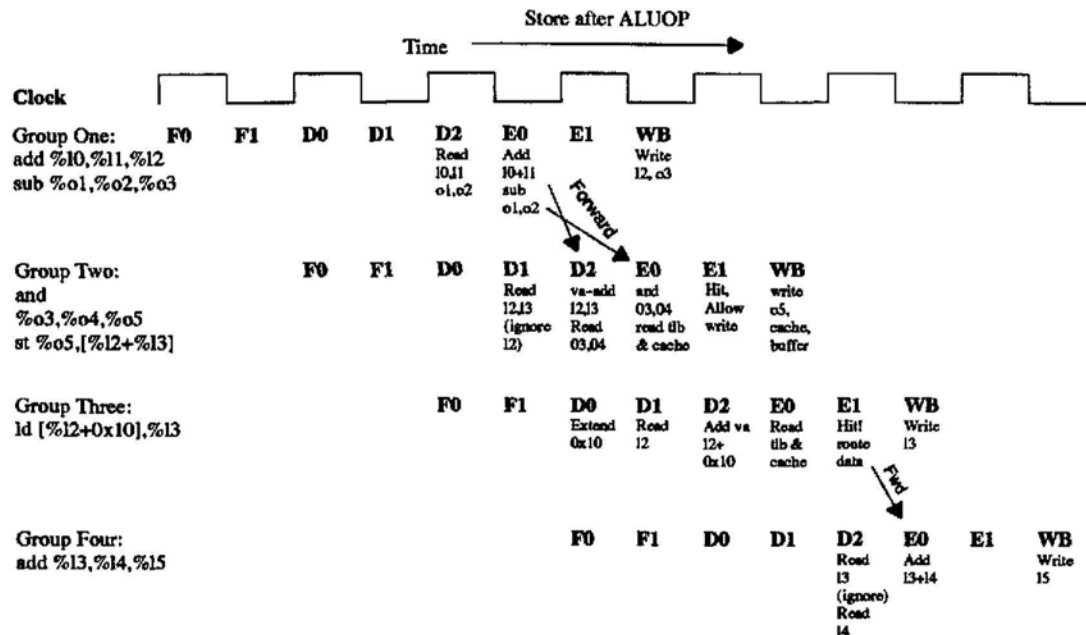
add    %10,%11,%12
sub    %01,%02,%03
!---Split (two write ports)
and    %03,%04,%05
st     %05, [%12+%13]
!---Split (only one memory reference)
ld     [%12+0x10],%13
!---Split (load data dependency)
add    %13,%14,%15

```

Note that the store in Example 5-3 requires data from the AND. This is executed with no delay, as the data to be stored is not required until the WB stage when it is actually written to the cache and/or memory.

Figure 5-3 is very similar to Example 5-2, with the addition of another memory reference instruction. Many of the forwarding paths used in the previous diagram are no longer used, while others are illustrated. The store address is computed in the D2 stage, then used to check the MMU and cache tags in E0/E1. Assuming all protection checks pass, the write is actually performed during WB.

Figure 5-3. Store Pipeline Operation



Operation of the data cache on an ST miss depends on whether SuperSPARC is operating directly on the MBus or is connected to the MultiCache Controller. When SuperSPARC is working with MXCC, if the store operation had missed the data cache, the timing would have been identical. The store data would have been written only to the store buffer and not to the data cache. In this case, SuperSPARC's data cache does not write-allocate on misses. For SuperSPARC directly on the MBus, however, if the store operation had missed the data cache, SuperSPARC's data cache would have brought the data from memory, retried the store (which would have hit by now), and then written the new data into the cache line. This process is called write allocation, and this new data would be copied back to memory when the cache line was replaced.

In some cases, a load needs to read recently written data. If this data is in the cache, it is returned immediately. For a SuperSPARC processor with MXCC, this data may not be in the cache, if the store missed in the internal data cache and the data are still in transit to the external cache or to main memory. The drain rate of the store buffer depends on the external system. In such cases, the store buffer is checked for a copy of the needed data; this is called "store buffer snooping." If the data is present, the processor requests that the store buffer be drained; this is called "store buffer copy-out." The processor waits until the requested data is no longer in the store buffer, and then reads the data back from memory, as in a normal load. SuperSPARC does not forward data from the store buffer to satisfy read requests. For SuperSPARC directly on the MBus, a store guarantees that the data cache has the new data. See Chapter 17 for more details.

## 5.5 Floating-Point Pipeline

Floating-point instructions utilize both the integer and floating-point units. SuperSPARC distinguishes between two types of floating-point instructions:

☐ **FPevs**

FPevs (floating-point events) include loads and stores into floating-point registers, are channeled directly from the integer unit to the floating-point registers.

☐ **FPOps**

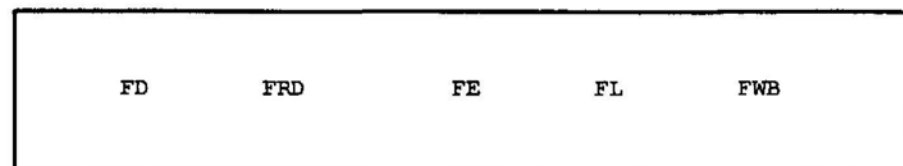
FPOps are placed in a four-entry-deep floating-point queue by the integer unit.

The floating-point unit reads the first instruction in this queue, performs the operation requested, and stores the results in its registers.

The four-entry floating-point queue enables SuperSPARC's IU to continue executing integer instructions while the floating-point unit takes care of the floating-point operations independently. If the integer unit tries to issue a floating-point operation to a full floating-point queue, the integer pipeline will stall until an entry is cleared from the queue.

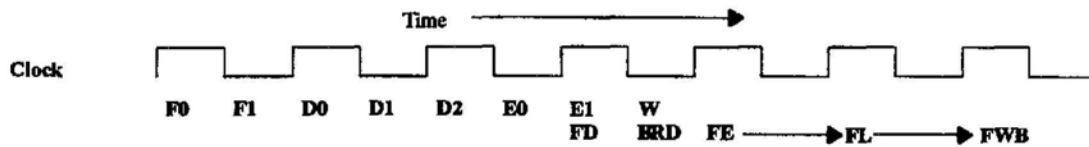
If a floating-point operation generates an exception, the floating-point unit will halt operation. The integer unit will be alerted to the trap on its next floating-point instruction issue. The floating-point exception trap handler should then store the contents of the queue before performing the trap. Each entry in the queue comprises a 32-bit FPop and its 32-bit virtual address. Once the trap is completed, this address enables the trap handler to reissue the stored instructions into the queue. In this fashion SuperSPARC is able to implement deferred floating-point traps. The floating-point queue is also called the floating-point deferred trap queue.

SuperSPARC's floating point pipeline is loosely coupled to the integer pipeline. A floating-point operation may be started every cycle. The latency of most floating-point instructions is three cycles. The FPU pipeline has the stages shown below. They are Decode, Read, Execute, Last, and Write-back.



Floating-point instructions are dispatched relatively late in the processor pipeline. They are not issued to the FPU until the E0 stage of the integer pipeline. Once issued, the floating-point instructions proceed through the FPU's pipeline, with FD occurring simultaneously with E1 in the integer pipeline, as in Figure 5-4. Forwarding paths are provided to chain the result of one floating-point operation to a source of a subsequent operation.

Figure 5-4. Floating Point Pipeline



In several cases, the floating-point pipeline becomes visible to the integer pipeline. When a branch on floating-point condition codes (FBfcc) instruction is issued, the processor may need to wait until a preceding FCMF instruction has completed. When a floating-point store instruction is executed, the pipeline also becomes visible. The integer pipeline waits in the E0/E1 stage until the requested data is available from the FPU.

In some cases, the floating-point queue may become full, typically when many long-latency floating-point instructions (e.g., divide and square root) or highly dependent operations are issued. If the floating-point queue becomes full, the integer pipeline will stall in E0 and wait for a queue entry to be freed by the completion of a FPop.

Since floating-point instructions are issued late in the pipeline (E0), and the actual arithmetic is not begun until one cycle later (WB), SuperSPARC may issue a load and a dependent FPop simultaneously. This is demonstrated in Example 5-4.

**Example 5-4. Floating-Point Pipeline**

```

ldd    [%l0],%f2
fadd    %f2,%f0,%f6
add     %l0,0x8,%l0
!--- Split (Three instruction max)
ldd     [%l0],%f4
add     %l0,0x4,%l0
fmuld   %f4,%f0,%f8
!--- Split (Three instruction max)
ldd     [%l0+4],%f10
cmp     %l0,0x100
be      Loop
!--- Split (Branch, Three instructions)
fadd    %f6,%f8,%f0
.

```

Example 5-4 shows many of SuperSPARC's strengths. All but the last group contain three instructions. The floating-point operations are grouped with load instructions that they depend on. The data returns from SuperSPARC's data cache at the end of the E1 stage; it is immediately used by the FPU's FRD stage and then by its FE stage.

Figure 5-5 shows some of the code in Example 5-5 being executed.

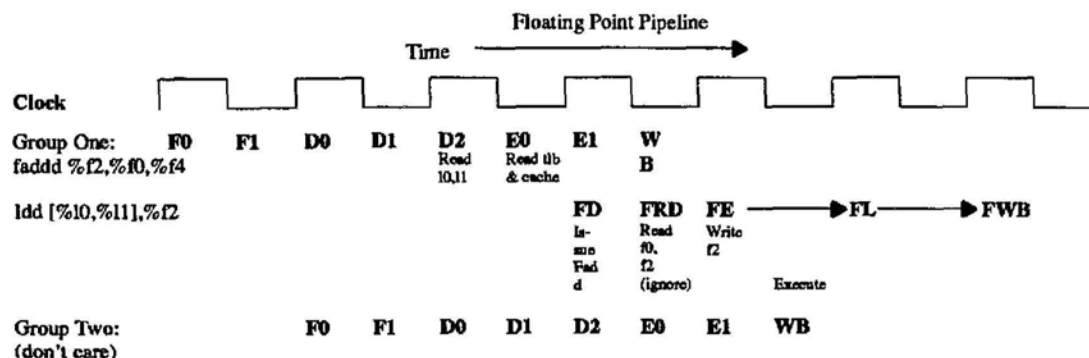
**Example 5-5. A Floating Point Pipeline Example**

```

ldd     [%l0+%l1],%f2
fadd    %f2,%f0,%f4
!--- Break

```

Figure 5-5. Floating-Point Pipeline Diagram



### 5.5.1 Floating-Point Instructions

There are two types of floating-point instructions:

- ☐ FPop (floating-point operations).
- ☐ FPEv (floating-point events).

The FPEvs (floating-point events) are executed by the IU and FPU together, but they do not enter the floating-point queue (FQ); these include:

- ☐ LDF/STF (load and store to floating-point registers).
- ☐ LDFSR/STFSR (load and store to floating-point status register).
- ☐ STDFQ (store floating-point queue).
- ☐ IMUL/IDIV (integer multiply and integer divide operations).

The FPOps include all FBfcc instructions, floating-point arithmetic instructions, and all floating-point moves, compares, and converts. These instructions all specifically enter the floating-point queue.

### 5.5.2 Floating-Point Queue

The FQ is a FIFO queue that holds a maximum of four FPOps. Should the queue become full, the integer pipeline of a group containing a FPop will stall until an entry clears. FSR.qne is reset (equals zero) in the FPU status register to indicate that the queue is empty. FPEvs do not enter the floating-point queue.

The contents of the FQ should be stored when the FPU goes into exception mode. This enables SuperSPARC to implement resumable, deferred floating-point exceptions. Any exception will remain pending until another floating-point operation is requested.

IDIV and IMUL can execute only in exception mode or when the FQ is empty. Normal floating-point operations will not resume until the integer multiply or divide completes.

### **5.5.3 Floating-Point Execution Times**

Most floating-point instructions have a three-cycle latency. More complicated instructions take longer; see Section 11.3 for more details.



## 5.6 Conditional Branches In the Pipeline

The handling of branches is critical to a reduced instruction set computer (RISC) processor's performance. SuperSPARC's branch implementation can execute both taken and untaken branches efficiently.

Untaken branches have a slight performance edge. The peak performance through an untaken branch is 3 instructions per cycle, while peak performance through a taken branch is 2.3 instructions per cycle. The difference comes from the delayed branch instruction, which, in the taken case, must be executed as a single-instruction group. Due to usual code scheduling restrictions, this peak performance will not generally be attained, and, in practice, the difference between taken and untaken branches is less.

SuperSPARC implements branch prediction; it always attempts to fetch the target of the branch. The pipeline assumes, however, that the branch is untaken, and it relies on the instruction queue having several sequential instructions available. The instructions at the branch target are fetched into a special buffer, the branch target queue. Once the direction of the branch has been resolved, the appropriate instructions are executed, either from the prefetch queue or the branch target queue.

When a branch instruction is encountered, the pipeline continues to execute normally for one cycle. In this additional cycle, the delay instruction is executed, possibly along with other instructions in the untaken stream. If the branch is taken, these instructions from the untaken stream should not have begun execution. SuperSPARC terminates the execution of these instructions for a taken branch as soon as the branch sequence has been resolved. This is called *aborting* the instructions. These instructions must be canceled before any machine state is modified.

A compare followed by a branch is a typical branch sequence. SuperSPARC executes this typical branch sequence by grouping together the branch instruction and one or more previous instruction(s). The delay instruction is executed in the next group. A typical instruction sequence might be as in Example 5-6 is a typical instruction sequence.

*Example 5-6. Branch Sequence*

```

subcc %l0,0x100,%g0
bz    TakenTarg
add   %l1,%l2,%l3
st    %l3,[%g1+0x100]
and   %o0,%o1,%o2
.
.
TakenTarg: st    %g0,[%g1+0x100]

```

This same code sequence can demonstrate both taken and untaken branches. The same control transfer mechanism can illustrate all forms of branches, jumps, and calls.

**5.6.1 Untaken Branch**

the sequence from Example 5-6 is shown in Example 5-7 with grouping for the untaken branch case.

*Example 5-7. Untaken Branch*

```

subcc %l0,0x100,%g0
bz    TakenTarg
!---Split (Split after CTI)
add   %l1,%l2,%l3
st    %l3,[%g1+0x100]
!---Split (No more write ports)
and   %o0,%o1,%o2

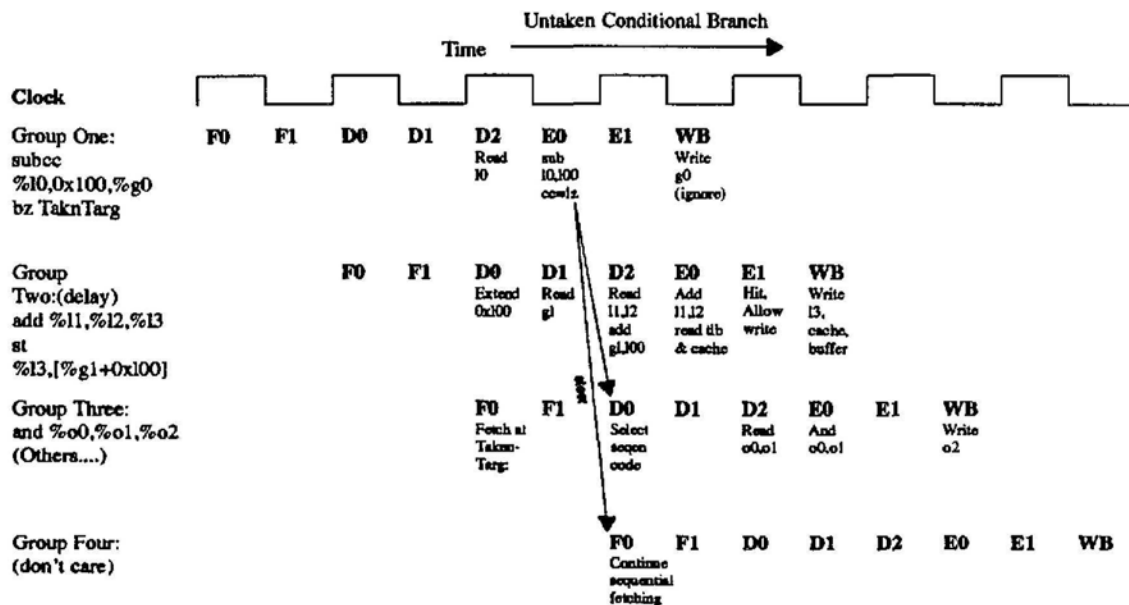
TakenTarg: st    %g0,[%g1+0x100]

```

The instructions at the label TakenTarg will not be executed, since the branch is not taken. Notice that two instructions (add, store) are executed in the delayed instruction group. The instructions beyond the third group are not significant for the example.

Figure 5-6 shows the fetch performed because the branch is assumed to be taken. The condition codes (from the subcc instruction) are resolved in the branch group's E0 pipeline stage. This condition code is used immediately to determine the branch direction in the delay group's D2 stage, the target group's D0 stage (by selecting the correct queue), and the following group's F0 stage (by selecting the correct prefetch program counter). Thus the branch is resolved after one cycle of uncertainty. Fetches once again follow the correct execution stream.

Figure 5-6. Untaken Branch Pipeline



## 5.6.2 Taken Branch

The taken branch case is more complicated. The most significant change is that instructions that begin execution in the delayed instruction group are aborted and never complete execution. Aborted instructions are boxed in the pipeline diagrams. The apparent grouping of instructions in the taken case is shown in Example 5-8.

## Example 5-8. Taken Branch Case Instruction Grouping

```

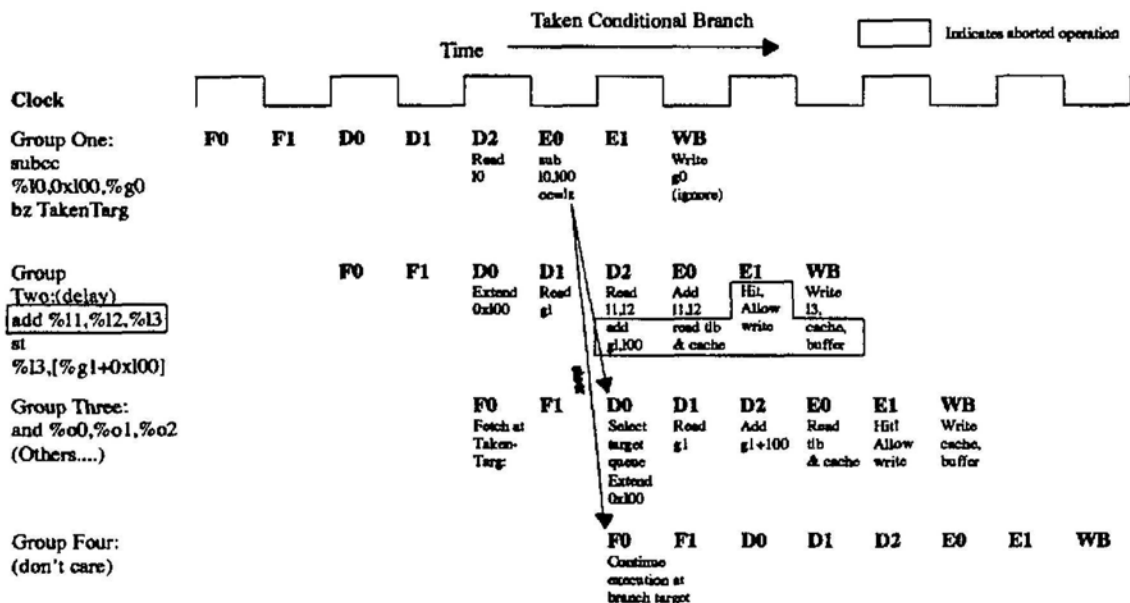
subcc %l0,0x100,%g0
bz    TakenTarg
!---Split (Split after CTI)
add    %l1,%l2,%l3
!---Split (abort st%l3, [%g1+0x100])

TakenTarg: st    %g0, [%g1+0x100]

```

The instruction grouping is the same as for the untaken branch case, up to the delay group. In the delay group, the grouping begins the same as for an untaken branch, but the grouping is modified by aborting the store instruction when the branch is resolved as taken. The taken branch pipeline is shown in Figure 5-7.

Figure 5-7. Taken Branch Pipeline



## 5.6.3 Branch Couple

SPARC allows for a limited set of Control Transfer Instruction (CTI) couples. They execute similarly to normal branches. SuperSPARC executes all the legal CTI couples correctly and, in conformance with SPARC V.8, does not support the implementation-dependent cases in the SPARC V.8 Specification.

#### **5.6.4 JMPL**

The JMPL instruction is an unconditional branch that uses a register as the source for the branch target. Other branch instructions compute the target address from an immediate operand and the current program counter value. JMPL requires an extra cycle because it has to read the register file before issuing a branch target fetch. The extra cycle is injected into the pipeline after the delay instruction of the JMPL (i.e., before execution of the target instruction).

## 5.7 Procedure Call and Return

SPARC defines primitive operations for calling procedures and returning from them under a variety of circumstances and language models. The CALL instruction is a primitive that branches to a subroutine, saving the value of the PC in a register. The JMWPL instruction branches to an address in a register and can be used for subroutine return.

Register windows are managed with the SAVE and RESTORE instruction. SAVE allocates a new register window, saving the previous one. The RESTORE instruction deallocates a register window and restores the previous window. SAVE and RESTORE can be used in the procedure call and return sequences, respectively. Both instructions also perform an addition that can be used for other functions in the calling and returning sequences, such as updating the stack frame pointer.

### 5.7.1 CALL, SAVE and RESTORE

CALL executes in the same manner as a taken branch, including execution of the delay instruction, except that the current PC value is written into register r15 (%o7). The value is written in the E0 stage of the group containing the CALL instruction. CALL is grouped with previous instructions in the same manner as branches. It does, however, require a register file write port, and so the grouping may be somewhat constrained by available resources.

SAVE and RESTORE are each a single instruction groups. Each is identical to an ADD instruction, except for the changing of CWP during the decode phase. SAVE and RESTORE require single cycles to execute.

### 5.7.2 Current Window Pointer (CWP) Pipeline

The SAVE instruction decrements the CWP by 1, while the RESTORE instructions increments the processor's CWP by 1. These instructions also perform an add operation. The sources for this add are from the old register window, while the destination is in the new window. For simplicity, SuperSPARC executes both these instructions as single-instruction groups. To make instructions before and after SAVE and RESTORE operations reference the correct windows, multiple current window pointers (CWPs) are maintained in the pipeline. Depending on an instruction's position relative to the CWP change, the appropriate CWP value is chosen.

---

**Note:**

Any manual modification of the CWP (using WRPSR) should be made in accordance with *The SPARC Architecture Manual* delayed write requirements. Several cycles may be required for a CWP update to take effect.

---

## 5.8 Exceptions and the Pipeline

Exceptions in the pipeline must ensure that all instructions before the exception complete while the instruction causing the exception and those after it (both in its group and in the following groups) do not.

In Example 5-9 the two arithmetic instructions form a cascade. The LDDF instruction is between the two arithmetic instructions. Also, the destination registers of the two arithmetic instructions are identical. The problem arises in this case if the LDDF instruction induces an exception due to a page-protection violation or a misaligned memory address. In this case, the first instruction (ADD) must complete and write back its result. The second instruction (LDD), however, must not complete.

### Example 5-9. Partial Execution of a Group of Instructions

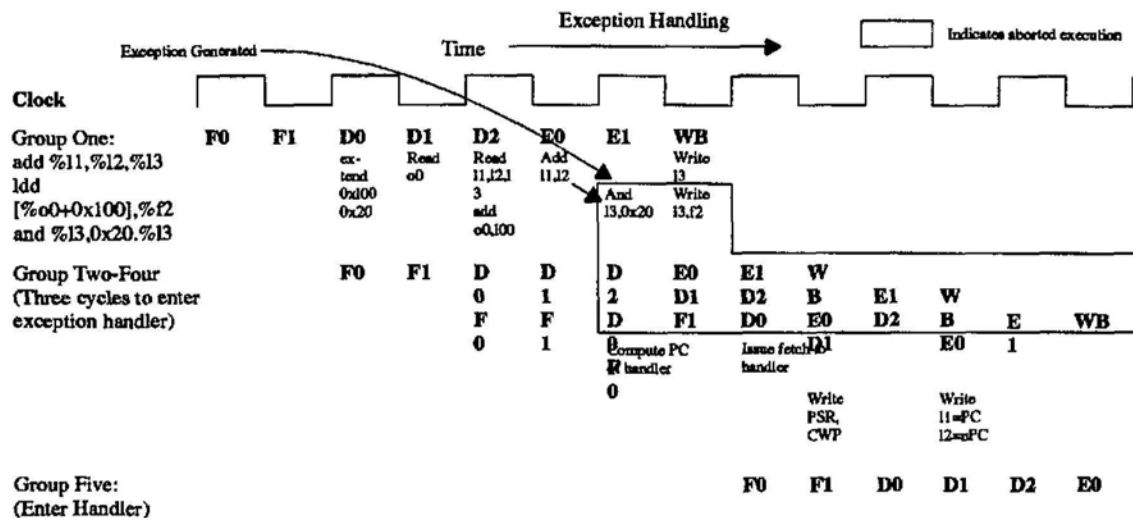
```
!--- Split (Start of group)
      add %l1,%l2,%l3
      ldd [%o0+0x100],%f2
      and %l3,0x20,%l3
!--- Split (3 instructions)
```

If the LDDF instruction had completed without incurring an exception, the result of the ADD would not have been written to the register file—it would simply have been used as a temporary value passed on to the AND. When the exception occurs, this is no longer true, and the result must be written to the register file.

### 5.8.1 Exception Pipeline

Example 5-9, with some surrounding instructions, is used in Figure 5-8 to illustrate the exception-handling pipeline. In Figure 5-8 an exception is reported for the LDDF instruction. The exception forces the ADD instruction to write its result and prevents the AND from doing so. The exception-handling logic then takes control of the pipeline. All subsequent instructions in the pipeline are aborted. The store buffer is copied out to memory, and the pipeline stalls until the copy-out is complete. In the next cycles, the PSR and CWP are modified to reflect the exception state of the processor. An instruction fetch to the proper interrupt handler in the trap table is requested. The exception PC and nPC are written into r17 and r18 (%l1,%l2). Finally, the instructions in the handler begin execution.

Figure 5-8. Exception-Handling Pipeline



## 5.8.2 Interrupts

Interrupts are handled in the same manner as exceptions. The interrupt request pins are sampled on the rising edge of the clock. In the next cycle, the highest priority interrupt is selected. This may be from either the interrupt request pins or from an internally generated interrupt (breakpoints). The request level of the interrupt is compared against the enable trap field and processor interrupt level field in the PSR (PSR.ET and PSR.PIL). If the interrupt request level (IRL) is higher than PSR.PIL, the instruction group at the E0 pipeline stage will, in the next cycle, trap to the interrupt handler.

For the interrupt to be accepted, there must be a valid instruction in the E0 stage. If there is no valid instruction, the interrupt is not taken until an instruction arrives at E0. This ensures that there is a valid PC to report as the interrupted program counter.

In a multiple-instruction group, the interrupt is reported to the last valid instruction in the group. This instruction is then aborted. Breakpoints are reported not to the last valid instruction but to the instruction that caused the breakpoint detection. The interrupt pipeline is identical to the exception pipeline (see Figure 5-8). It behaves just as if an exception had been reported to the last valid instruction in the E0 group.



### 5.8.3 Return From Trap (RETT) Pipeline

The return from trap instruction executes in cooperation with the JMPL that must immediately precede it. (See *The SPARC Architecture Manual*). Execution of a RETT is similar to a JMPL but also updates current window pointer (CWP) and restores S and ET. The JMPL returns to the delay instruction, and the RETT returns to the target of the branch. If there was no previous branch, NPC is set to PC+4.

A RETT executes as a single instruction group. It incurs no additional pipeline cycles. The preceding JMPL introduces a bubble into the pipeline after the RETT.

The two-instruction pair is needed because the trapping instruction might have been the delay instruction of a delayed control transfer operation (DCTI). In that case, the NPC would not be PC+4 and would have to be restored from the register containing the saved nPC.

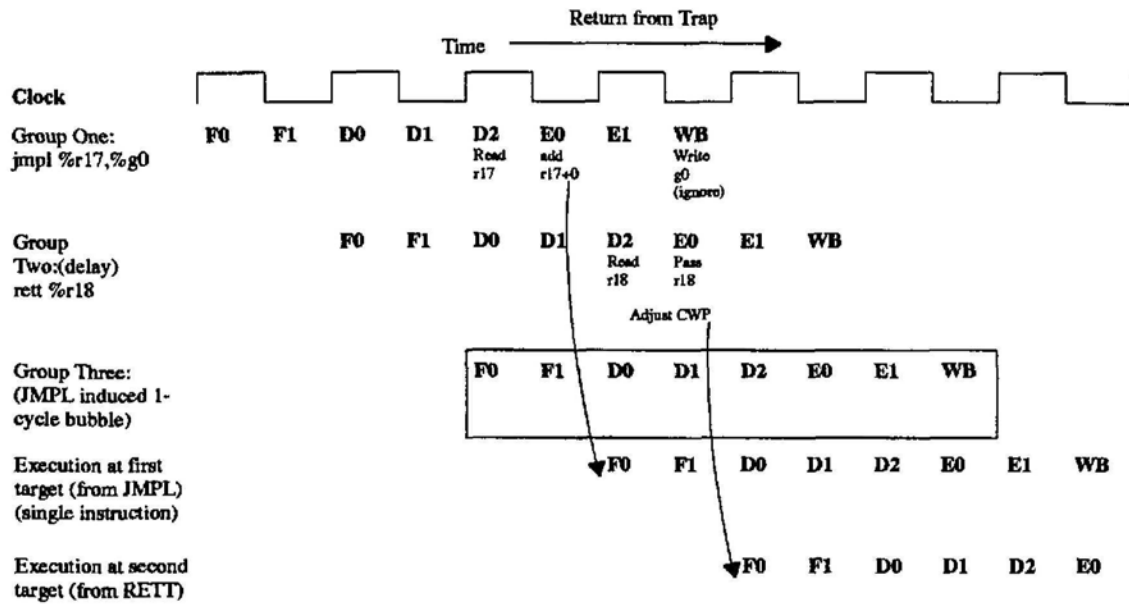
Example 5-10 illustrates the usual code sequence used to return from a trap.

*Example 5-10. Return From Trap*

```
    jmp1    %r17,%g0
    rett    %r18
```

The instructions at the target of the JMPL are executed as a single instruction group, since it may not be contiguous with the return address in the RETT. The pipeline operation for the sequence is shown in Figure 5-9.

Figure 5–9. Return From Trap Pipeline





# **Code-Generation Principles**

---

---

The performance available from the SuperSPARC processor (SSP) can vary significantly with the actual instruction sequence of the program. This chapter presents a number of guidelines that assembly language programmers and compiler code generators can use to increase the performance of programs on the SSP. The rules used by the SSP to group instructions are also presented.

<b>Topic</b>	<b>Page</b>
<b>6.1 Performance Limiters .....</b>	<b>6-2</b>
<b>6.2 Performance of Existing Code .....</b>	<b>6-3</b>
<b>6.3 Resource Allocation .....</b>	<b>6-4</b>
<b>6.4 Spreading the Use of Critical Resources .....</b>	<b>6-6</b>
<b>6.5 Code-Generation Guidelines .....</b>	<b>6-7</b>
<b>6.6 Instruction Grouping .....</b>	<b>6-16</b>

## 6.1 Performance Limiters

The performance of a program running on the SSP depends on how the program's instructions are formed into groups for superscalar processing. The ordering of the instructions in the program can greatly impact the way they are grouped and thus the number of cycles used to complete the program. The compiler code generator or assembly language programmer can plan or schedule the instructions to make the best use of the SSP's internal resources and achieve greater performance.

The performance of programs on the SSP is sensitive to many factors. The significant performance limiters vary greatly between programs. Classes of limiters include:

- ☐ Branch frequency and direction.
- ☐ Memory-reference patterns.
- ☐ Floating-point operation scheduling.
- ☐ Instruction ordering.
- ☐ Data and register dependencies.

In floating-point programs, branch performance is rarely a limiter. This is because loops are often unrolled. In most cases, floating-point programs are memory-reference limited rather than arithmetic-limited. Since only a single memory reference can be executed per instruction group, interleaving memory references with nearly anything else improves performance.

Most programs are limited by branch performance or load/store bandwidth. In particular, a taken branch can execute only a single instruction in the branch delay group. Load and store operations are often bunched together. This prevents other instructions from being executed in parallel with the memory references.

Cache performance is also critical to achieving maximum performance. Many routines are small enough for their entire code and data sets to be contained in the SSP's on-chip caches. Cold-start penalties, which are usually insignificant, will degrade the performance of such routines. Most larger programs, however, do not fit entirely in the on-chip caches. These programs typically lose 20% to 30% of their performance to cache miss penalties. Prefetch logic, in some cases, will limit the decrease in performance. Localizing code and data references within a program can lead to substantial performance improvements.

## **6.2 Performance of Existing Code**

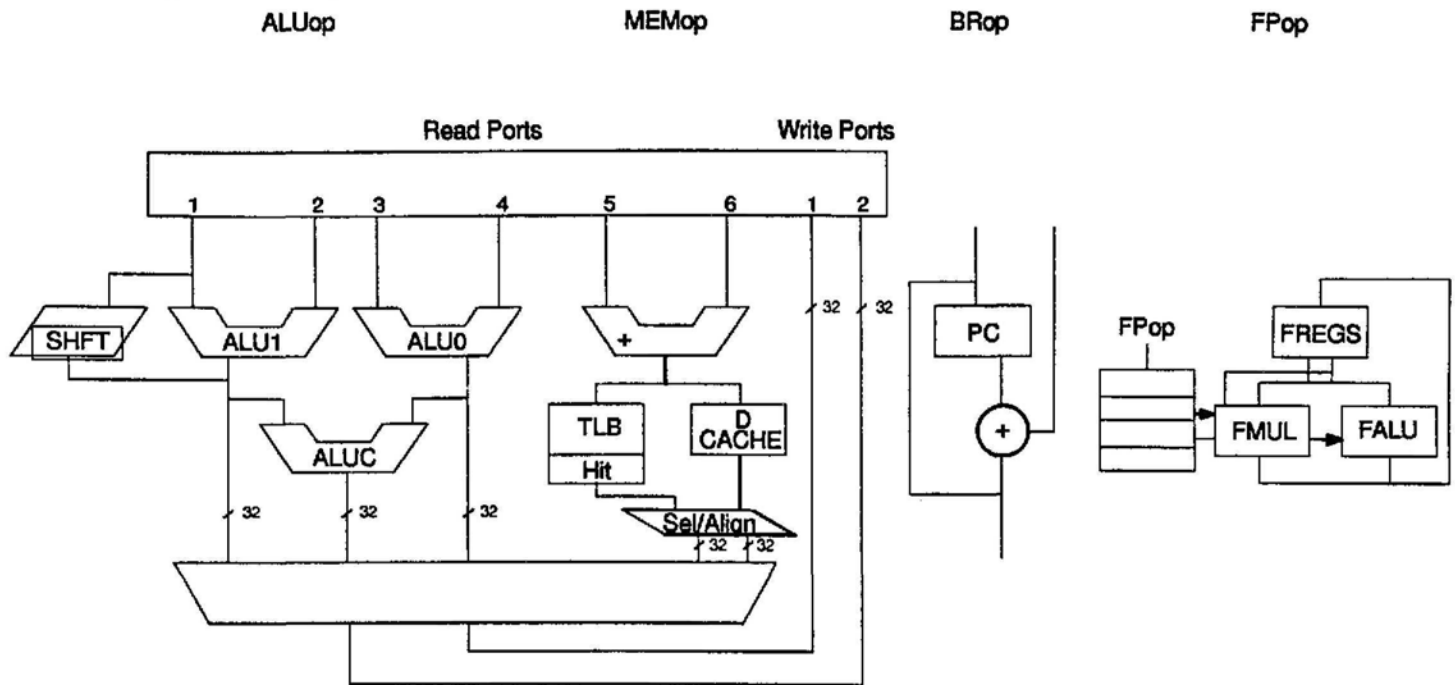
The SSP is designed to execute code from older SPARC compilers efficiently. In general, the SSP executes this code at between 1.4 and 1.6 instructions per cycle (IPC), or 0.6 to 0.7 cycles per instruction (CPI). This decreases to about 1.1 IPC for large programs not fully contained in the cache. The performance of floating-point programs is often greater.

This performance level can be increased significantly with code generated explicitly for the SSP. The remainder of this chapter considers low-level code-scheduling issues when generating code to run on the SSP.

### 6.3 Resource Allocation

The SSP must group instructions according to available hardware resources. The hardware resources and restrictions are described in Figure 6-1.

Figure 6-1. SuperSPARC Processor Hardware Resources and Restrictions



#### 6.3.1 Ports to Memory

The most basic hardware resource limitation is a single port to memory. This is what restricts the SSP to one load or store per cycle. Similarly, a single port to the instruction cache restricts branch performance.

#### 6.3.2 Ports to the FPU

Although the SSP has an independent floating-point adder and multiplier, only one floating-point operation per cycle can be dispatched. A load or store between a floating-point register and memory, however, can be done in the same cycle as a floating-point operation.

#### 6.3.3 Integer Register Write Ports

Two write ports to the integer register file are available. All arithmetic instructions use one of these ports. Load instructions use one write port, and load double instructions use both write ports. Note that floating-point load operations do not use any integer register write ports. Example 6-1 is a code segment that loads double floating-point registers and executes in two cycles.

*Example 6–1. Integer Register Write Ports—Double Floating-Point Registers*

```

add    %l0,%l1,%l2
and    %l2,%0xff,%l3
ldd    [%o0+0x100],%f2
!---- split (Three instructions)
add    %l4,%l5,%l6
and    %l6,0xff,%l7
ldd    [%o0+0x108],%f4
!---- split (Three instructions)

```

A write port is used when the %g0 register is used as a destination because the condition codes must be carried along with the instruction. A write port is also used by a store instruction.

Example 6–2 is similar to Example 6–1 but loads into integer registers and executes in four cycles.

*Example 6–2. Integer Register Write Ports—Integer Registers*

```

add    %l0,%l1,%l2
and    %l2,%0xff,%l3
!---- split (No more write ports)
ldd    [%o0+0x100],%i2
!---- split (No more write ports)
add    %l4,%l5,%l6
and    %l6,0xff,%l7
!---- split (No more write ports)
ldd    [%o0+0x108],%i4

```

**6.3.4 Integer Arithmetic Units**

The SSP can perform up to two arithmetic logic unit operations (ALUops) in an instruction group. This is implemented internally with three separate ALUs and a shifter. Each of these can produce one result per cycle. One of the arithmetic logic units (ALUs) can use the output of either the other ALU or the shifter as its input.

The number of ALUs cannot limit the machine's performance, since no more than two results can be stored by the register file. Only a single shifter exists, however, which means that a result cannot be cascaded into a shift operation.

The shift result is produced early enough for it to be the source of a cascaded operation. This allows for such common combinations of operations as shift&add and shift&compare.



## 6.4 Spreading the Use of Critical Resources

A single simple guideline can assist greatly in generating code that performs well on the SSP: interleave as many different classes of operations as possible, spreading members of the same class as far as possible from each other. A simple measure of the minimum execution time attainable for a routine is shown in Equation 6-1.

Equation 6-1. Minimum Execution Cycles

$$\text{Min Cycles} = \text{Max} \left\{ \begin{array}{l} \text{Memory References} \\ \text{Floating-Point Operations} \\ \text{Integer Operations} \\ \text{Branch Operations} \times 2 \end{array} \right.$$

The rule holds when one, and possibly two, of the terms are close to the maximum. As more terms get larger, the likelihood of achieving or approaching the minimum number of cycles decreases.

In order to approach the minimum above, all of the classes of operations must be interleaved as much as possible. In the worst case, the maximum number of execution cycles is shown in Equation 6-2.

Equation 6-2. Maximum Execution Cycles

$$\begin{aligned} \text{MaxCycles} = & \text{Memory References} + \text{Floating-Point Operations} \\ & + \left( \frac{\text{Integer Operations}}{2} \right) + (\text{Branch Operations} \times 2) \end{aligned}$$

## 6.5 Code-Generation Guidelines

The SSP's dynamic grouping logic compensates locally for minor scheduling variations. The sequence in Example 6-3 executes in the same number of cycles as the sequence in Example 6-4.

*Example 6-3. Code Scheduling Example 1*

```
ld    [%10],%g1
!--- split (only one memory reference)
ld    [%11],%g4
add    %g1,0x100,%g3
!--- split
```

*Example 6-4. Code Scheduling Example 2*

```
ld    [%10],%g1
!--- split (dependent Ld-Use)
add    %g1,0x100,%g3
ld    [%11],%g4
```

Example 6-4 demonstrates the self-aligning nature of execution on the SSP. The pipeline aligns instruction groups based on the positions of the critical operations in the code (the memory references, floating-point operations (FPops), and branches).

The important consideration is that the critical operations do not become serial but rather are interleaved to increase parallel operation. Most sequences of instructions have many optimal schedules.

### 6.5.1 Reduction of Branches

As shown in Equation 6-1 and Equation 6-2, each branch tends to increase the execution time of a sequence by about two cycles. Thus it is more significant to remove a branch than to remove a memory reference.

Code should be unrolled wherever possible. This is a performance boost on most machines, especially on the SSP. The addition of as many as four other instructions is generally better than a single branch. Where possible, arithmetic logic, rather than sequences of branches, will improve performance.

### 6.5.2 Allocation of Delay Instructions

Since the delay instruction of a taken branch is forced to be a single instruction group, it is very important to make the best use of that instruction. For example, in unrolled LINPACK, optimal performance cannot be achieved unless a memory reference is placed in the delay instruction.

For maximum performance, the delay instruction should be used to execute one of the critical performance-limiting operations in the code.

Reorganizing code to properly fill the delay instruction may require adding instructions to the code. This is a trade-off that should be made carefully. Even though adding instructions can increase performance on the SSP, it will decrease performance on any non-superscalar machine. In addition, it expands the size of the program, which may increase the number of instruction cache misses. One alternative is to use software pipelining, which often allows the same performance levels without increasing the number of instructions. This is accomplished by spreading the execution of each loop iteration over several actual trips through the code. The resulting code in one loop would execute the final operations of the previous loop, the core of the current loop, and the prologue of the next loop.

Annulled branches should be used only if there is no alternative. An annulled branch saves only the code space used for the delay instruction. The cycle in which the delay instruction of the annulled branch would have been executed is still required.

### **6.5.3 Reduction of Floating-Point Register Dependencies**

Floating-point register dependencies can be significant performance limiters. No floating-point register can be modified until all pending operations in the floating-point queue that either produce or use that register have completed. The Floating-Point Unit (FPU) has a fairly long execution pipeline, and performance suffers when there are frequent FP data dependencies. The SSP's performance may be increased by using more loads and stores to move floating-point data back and forth between memory and registers, as opposed to overusing a single register and introducing dependencies.

Note that these guidelines do not apply to integer operations. Integer operations have much shorter latency, and integer register dependencies are handled more effectively. Example 6-5 and Example 6-6 show bad and good examples, respectively, of FP register dependencies.

*Example 6–5. Bad Example of FP Register Dependencies*

```

ld    [%i0],%f1
!--- split (Only one memory reference)
ld    [%i0 + 4],%f0
fmuls %f0,%f1,%f0
!--- split (Only one memory reference)
ld    [%i0 + 8],%f1
fadds %f0,%f1,%f2
!--- split
!--- PIPELINE STALL FOR 2 CYCLES!
!--- until fmuls produces %f0
!--- finishes with %f1

```

Notice that the last load operation reuses register %f1. This prevents the load operation from being executed until after the FMULS has completed. The main pipeline will stall for two cycles. A better schedule is shown in Example 6–6.

*Example 6–6. Good Example of FP Register Dependencies*

```

ld    [%i0],%f1
!--- split (Only one memory reference)
ld    [%i0 + 4],%f0
fmuls %f0,%f1,%f0
!--- split (Only one memory reference)
ld    [%i0 + 8],%f3
fadds %f0,%f3,%f2
!--- split

```

By changing the register used in the last load and add operations, the two pipeline stalls are removed completely.

The FADDS uses %f0 as a source operand and can only start execution when the FMULS has produced a result for %f0. The FADDS is issued into the FQ, however, and the main pipeline does not stall. Another optimization that may be applied to this code is using a load double to replace the first two memory references. Such optimizations must be done carefully.

**6.5.4 Other Floating-Point Code Issues**

The following section provides examples of floating-point codes that illustrate the SSP's FPU operation. Each example is preceded by a brief explanation about the significant details.

### 6.5.4.1 Throughput

Example 6-7 shows how to take full advantage of FPop latency (for example, three-cycle latency of FADDs) and execute each FPop without any pipeline stall. This arrangement achieves the highest throughput.

#### Example 6-7. Throughput Example

```
fadds %f0,%f1,%f20
!--- split -----
fadds %f2,%f3,%f21
!--- split -----
fadds %f4,%f5,%f22
!--- split -----
fadds %f6,%f7,%f23
st    %f20,[%l1]
!--- split -----
fadds %f8,%f9,%f24
.....
```

### 6.5.4.2 One Stall

Example 6-8 is similar to the previous code sequence, except that the ST is attempted a cycle earlier. Because of the three-cycle required latency of FADDs, a bubble will be inserted to stall the pipeline before the ST can be completed. The SSP's grouping logic places FADDs %f4,%f5,%f22 together with ST %f21,[%l1] in the same group (this is done prior to the FPU detecting any dependency). Therefore, when ST is stalled, the group remains the same.

#### Example 6-8. One-Stall Example

```
fadds f0,%f1,%f20
!--- split -----
fadds %f2,%f3,%f21
!--- split -----
fadds %f4,%f5,%f22    (issued)
st    %f21,[%l1]      (issued)
!--- Split-----
stall
!--- split -----
!(fadds %f4,%f5,%f22  completed)
!(st    f21,[%l1]     completed)
!--- split -----
```

### 6.5.4.3 FCMP/FBfcc Latency

Example 6-9 demonstrates the latency of an FCMP-FBfcc pair. An FCMP requires three cycles before the floating-point condition codes are resolved for an FBfcc use. Arranging the code as given in Example 6-9 allows each cycle to complete without any pipeline stall.

Example 6-9. FCmp/FBfcc Latency

```
!--- split -----
fcmps %f6,%f7
!--- split -----
fadds %f0,%f1,%f21
!--- split -----
fadds %f2,%f3,%f22
!--- split -----
fadds %f4,%f5,%f23
!--- split -----
fbne t1
!--- split -----
```

### 6.5.4.4 FCMP/FBfcc Stall

The FPU detects whether there is a pending FCMP and stalls the pipeline if it encounters an FBfcc that may need the FCMP's result. In Example 6-10, an FBfcc is issued immediately after an FCMP. The SSP issues the FCMP but then, before executing the FBNE, it stalls the pipeline for three cycles until the FCMP has completed.

Example 6-10. FCmp/FBfcc Stall

```
!--- split -----
fcmps %f6,%f7 (issued)
!--- split -----
stall
!--- split -----
stall
!--- split -----
stall
!(fcmps %f6,%f7 completed)
!--- split -----
fbne t1
!--- split -----
```

#### 6.5.4.5 f Register Dependency on Store

Example 6-11 shows how a floating-point register dependency with a store may stall the pipeline. The SSP groups the FADDs, ST, and CMP instructions together and then begins execution. The FADDs instruction requires three cycles to compute results, thus stalling the integer pipeline for three cycles before these instructions can complete.

Example 6-11. f Register Dependency on Store

```

!---- split -----
fadds %f0,%f1,%f21 (issued)
st    %f21,[%l1]    (issued)
cmp    %i7,2        (issued)
!---- split -----
stall
!---- split -----
stall
!---- split -----
stall
! (fadds %f0,%f1,%f21 completed)
! (st    %f21,[%l1]    completed)
! (cmp    %i7,2        completed)
!---- split -----

```

Note in Example 6-11 that a floating-point exception from FADDs will be reported to the ST. `fp_exception` is a deferred trap and is always reported to the next FPop or FPev.

#### 6.5.4.6 f Register Dependency on Load

Example 6-12 shows how a floating-point register dependency before a load may stall the pipe. The SSP groups the FADDs, LD, and the CMP, and then begins execution. The LD cannot complete before the FADDs instruction is completed, which takes three cycles; the pipeline is therefore stalled for that amount of time. This is required to avoid destroying the source registers for current FPop.

**Example 6–12. f Register Dependency on Load**

```

!--- split -----
fadds %f0,%f21,%f1 (issued)
ld    [%l1],%f21   (issued)
cmp    %i7,2        (issued)
!--- split -----
stall
!--- split -----
stall
!--- split -----
stall
! (fadds %f0,%f21,%f1 completed)
! (ld    [%l1],%f21   completed)
! (cmp    %i7,2        completed)
!--- split -----

```

Note that any floating-point exception from FADDs will be reported to the LD in Example 6–12.

**6.5.4.7 f Register Data Forwarding (Output to Input)**

Example 6–13 demonstrates how a floating-point register dependency between two FPOps activates the data forwarding and does not cause the pipeline to stall. The destination register of FADDs %f0, %f1, %f21 is a source register for FADDs %f21, %f2, %f22 (both FPOps are able to enter the FQ immediately), but the second FADDs will not start until the first FADDs reaches FWB stage. Since the second FADDs receives the data from the forwarding path, it does not wait until the first FADDs has written the data into the register.

**Example 6–13. f Register Data Forwarding (Output to Input)**

```

!--- split -----
fadds %f0,%f1,%f21
!--- split -----
fadds %f21,%f2,%f22
!--- split -----

```

**6.5.4.8 f Register Data Forwarding (Input to Output)**

Example 6–14 demonstrates that a floating-point register dependency between two FPOps does not cause either the integer or floating-point pipeline to stall. Both instructions are able to complete immediately.



**Example 6–14. f Register Data Forwarding (Input to Output)**

```

!--- split -----
fadds %f0,%f1,%f21
!--- split -----
fadds %f2,%f3,%f0
!--- split -----

```

**6.5.5 Spread Address Calculation and Memory Reference**

In order for the SSP to implement single-cycle memory references, the address registers must be stable by the D2 pipeline stage of the memory reference. This implies that a result for a register used for address computation must be completed no later than the E0 stage of the previous instruction group.

Example 6–15 illustrates that, since the results of the shifts are needed for the load address calculation, they must be executed in separate groups. If the shift had not been producing data for that load, it could have been grouped together with the load. With the address calculation dependency, however, the shift will be grouped with previous instructions, and the load will be grouped with subsequent operations.

**Example 6–15. Address Calculation Dependency on Cascade**

```

sll    %t0,0x3,%t0
!--- split (ALUOP into Memory reference Address)
ld     [%t0+%t0],%t1

```

The next case, as shown in Example 6–16, is less common but occurs when the address is calculated in a cascaded instruction group. The results of a cascade are not available until the end of the E1 execution stage. Since the memory reference requires data at the end of E0, it must include a pipeline bubble.

**Example 6–16. Spread Address Calculation and Memory Reference**

```

sll    %t0,0x3,%t0
add    %t0,%t0,%t0
!--- split (Cascade into memory reference)
bubble
!--- split -----
ld     [%t0],%t1

```

Note that Example 6-16 requires three cycles to execute rather than the two cycles required in Example 6-15. A sequence such as this is used rarely—only when complex address arithmetic is required.

A third case often occurs during linked list traversal and is illustrated in Example 6-17. A bubble is inserted when the results of one load are immediately used as part of the address calculation for the next load.

**Example 6-17. Indirect Address Calculation**

```
ld    [%10],%11
!--- split (load into memory reference)
bubble
!--- split
ld    [%11],%12
```

The pipeline stall is required for the same reason as in Example 6-16—the results of a load instruction are not available until the end of the E1 pipeline stage.

If any of the cases in Example 6-15, Example 6-16, and Example 6-17 can be spread apart by moving other instructions between these dependencies, performance will be increased.

## 6.6 Instruction Grouping

The SSP forms groups of instructions in the D0 pipeline stage by examining the available instructions from its instruction queue. A set of grouping rules decides which of the instructions will be selected for inclusion in the next group to be executed. The SSP will select no instructions, the first instruction, the first two instructions, or the first three instructions to form a group.

The size of the group is determined not only by the instructions of the program but also by the instructions actually available in the instruction queue. The instruction queue may contain fewer than three instructions due to branches and instruction cache misses.

When no instructions are available in a cycle, the grouping logic issues a zero-instruction group. Zero-instruction groups are called bubbles. Bubbles traverse the pipeline like other groups but do not execute instructions. A bubble cannot cause a pipeline hold but is subject to pipeline holds from other groups in execution.

There are two classes of grouping rules. The classes are the split after rules and the split before rules. These rules determine whether the instruction group will be terminated before or after a particular instruction. These two classes contain rules based on the available instructions, rules based on the previous instruction group, and rules based on exceptions.

The following sections provide descriptions of all these rules.

There are many grouping rules that are required to handle pipeline hold conditions. These rules are not listed here.

### 6.6.1 Split After Rules

The following rules split the group after an instruction based on relations among the first three instructions in the queue. These rules will prevent any further instructions from being included in the current group.

- ☐ Split after first valid exception.
- ☐ Split after any control transfer instruction.
- ☐ Split after condition codes set in cascade.
- ☐ Split after MULScc destination not equal to source of next MULScc.
- ☐ Split after first instruction after annulled branch.
- ☐ Split after first instruction midway through a branch couple.

#### 6.6.1.1 Split After First Valid Exception

The group is split after the first exception; this prevents instructions from entering the pipeline after an instruction\_access\_exception has been signaled. The exception will travel through the pipeline and actually only occur when the instruction generating the exception reaches the E0 stage of the pipeline.

### 6.6.1.2 Split After Any Control Transfer Instruction

This rule splits the current group between any branch and the delay instruction that follows the branch. Any instructions that were grouped along with a branch appeared before it in the program.

### 6.6.1.3 Split After Condition Codes Set in Cascade

This rule prevents any additional instructions from being accepted after an ALUop cascade in which the second ALU operation sets condition codes. See Example 6-18.

#### Example 6-18. Split After Condition Codes Set in Cascade

```
add    %l5,%l3,%g2
subcc  %g2,%g0,%g0
!---  split after cc set in cascade
bge    loop
```

### 6.6.1.4 Split After MULScc Destination Not Equal to Source of Next MULScc

This rule prevents multiple MULScc instructions from being executed in parallel unless the second instruction uses the result of the first instruction. The normal usage of MULScc has the destination equal to the source of the next MULScc. Other usage is uncommon, but it is architecturally legal.

### 6.6.1.5 Split After First Instruction After Annulled Branch

This rule prevents multiple instructions from executing in the delay group of an annulled branch. The instruction in the delay group of an annulled branch will normally be executed only if the branch is taken. When this occurs, only the first instruction after the branch is expected to be executed. If the previous branch is untaken, the instruction in the delay position is not executed at all. The SSP will begin executing this delay instruction, assuming the branch will be taken, and, if necessary, abort it later.

### 6.6.1.6 Split After First Instruction Midway Through a Branch Couple

This rule prevents multiple instructions from being executed when a branch couple is being processed. SPARC allows for a restricted number of branch couple conditions. The SSP will only execute single instructions during branch couples. Example 6-19 demonstrates this rule. The BE is executed in the delay slot for the BA and assuming the BE succeeds only the first instruction at dest1 should be executed before continuing execution at dest2.

*Example 6–19. Split After First Instruction Midway Through a Branch Couple*

```
ba dest1
!--- split after CTI
be dest2 !--- delay instruction
!--- split after CTI
(Never Executed)
.
dest1: add %10,%11,%12
!--- split midway through branch couple
add %13,%14,%15
.
dest2: add %14,%15,%16
```

### 6.6.2 Split Before Rules

Split before rules are typically used to split a group when a resource required by the candidate instruction is unavailable. For example, all instruction groups are split before a second memory reference.

The “Split Before” rules are as follows:

- ☐ Split before invalid instruction queue entry.
- ☐ Split before out of integer register read ports.
- ☐ Split before out of integer register write ports.
- ☐ Split before second memory reference.
- ☐ Split before second shift.
- ☐ Split before second Fpop.
- ☐ Split before second cascade.
- ☐ Split before cascade into shift.
- ☐ Split before cascade into JMPL.
- ☐ Split before cascade into memory reference address.
- ☐ Split before load data cascade use.
- ☐ Split before cascade in previous group into memory reference address.
- ☐ Split before sequential instruction.

- ☐ Split before control register read after previous Setcc.
- ☐ Split before MULSCC unless first one or two instructions.
- ☐ Split before extended arithmetic from cc set in current group.
- ☐ Split before delay group CTI unless first.
- ☐ Split before CTI in JMPL delay unless RETT.

#### **6.6.2.1 Split Before Invalid Instruction Queue Entry**

The SSP uses this rule to wait for instructions from the instruction queue and instruction cache. Instruction access exceptions are considered valid instruction queue entries so that they can proceed down the pipeline and be recognized. It is possible for fewer than three instructions to be valid from the queue. This rule limits the maximum number that can be executed at a given time to the instructions available in the instruction queue.

#### **6.6.2.2 Split Before Out of Integer Register Read Ports**

This rule prevents a group from using more register file read ports than are available. Four operand read ports are available. Memory address register ports are independent and do not affect this rule.

#### **6.6.2.3 Split Before Out of Integer Register Write Ports**

This rule prevents a group from using more register file write ports, condition code ports, or result forwarding resources than are available. There are two operand write ports.

Each ALUop (including SETHI and instructions sending a result to %g0) uses one write port. Load instructions up to word size use one write port. Load double integer (LDD) instructions use both write ports.

#### **6.6.2.4 Split Before Second Memory Reference**

The SSP has only a single port to memory. This rule prevents a group from attempting to use it twice and does not discriminate between floating-point or integer load and stores.

#### **6.6.2.5 Split Before Second Shift**

Although the SSP has several ALUs, only a single shifter is provided.

#### **6.6.2.6 Split Before Second FPop**

Only one FPop may be issued to the FPU in an instruction group.

#### **6.6.2.7 Split Before Second Cascade**

Only one operand may be cascaded into the cascade ALU in step E1 of the integer pipeline. This rule prevents two E0 results from being used to form a cascaded operation.

#### 6.6.2.8 Split Before Cascade Into Shift

The shifter must be used in the first execute stage and therefore may not be the second operation in a cascade.

#### 6.6.2.9 Split Before Cascade Into JMPL

JMPL reads from the register file to calculate the target of its branch. The value is required before the E0 stage. This requires that the register not be changed in the group with the JMPL. This rule detects the existence of such a condition and forces a split before the JMPL.

#### 6.6.2.10 Split Before Cascade Into Memory Reference Address

The register values used for address calculation for memory reference instructions may not be produced in the same group as the memory reference, as shown in Example 6-20. In order to perform single-cycle memory references, the registers forming the address for a load or store must be available before the D2 pipeline stage of the memory reference. This requires that they be changed no later than the E0 pipeline stage of the previous instruction group. This rule and the *Split Before Previous Cascade into Memory Reference Address* rule enforce this restriction.

#### Example 6-20. Split Before Cascade Into Memory Reference Address

```
add    %15,%13,%g2
!--- split before cascade into mem ref
ld      [%g2+0x10], %g1
```

#### 6.6.2.11 Split Before Load Data Cascade Use

A full cycle is required to access the on-chip data cache. The data from a load is available after the E1 stage of the load. Therefore, it may not be used before the E0 stage of the next instruction group. This rule prevents an instruction from using that data in the group with the load instruction (during E0 or E1).

#### 6.6.2.12 Split Before Previous Cascade Into Memory Reference Address

The registers used for address calculations for memory reference instructions must not have been generated in the cascade of the previous group. This rule enforces an extension of the Split Before Cascade into Memory Reference Address rule. It requires that the address registers stabilize by the end of the previous instruction group's E0 pipeline stage. In Example 6-21 the SUBcc and the LD would have been grouped together but will be split by this rule.

**Example 6–21. Split Before Previous Cascade Into Memory Reference Address**

```

add    %l5,%l3,%g2
add    %g2, %l2, %g4
!--- split (out of write ports)
subcc  %g4, 0x10, %g0
!--- split before cascade in prev group
ld     [%g4+0x10], %g1

```

**6.6.2.13 Split Before Sequential Instruction**

The SSP has a set of instructions that can only be executed as single instruction groups. In addition, the SSP can be forced to execute all instructions as single instruction groups through an ASI visible control bit (ACTION.MIX). See Subsection 15.2.5.

The set of instructions that are always single instruction groups is as follows:

- ☐ SAVE and RESTORE.
- ☐ LDD and STD operations in integer registers.
- ☐ All alternate space stores: STA, STDA, STBA, STHA.
- ☐ Atomic operations: SWAP, LDSTUB.
- ☐ All control register accesses: read and write PSR, ASR, WIM, Y.
- ☐ RETT.
- ☐ FLUSH.
- ☐ All software traps: Ticc.
- ☐ Integer multiply: UMUL, UMULcc, SMUL, SMULcc.
- ☐ Integer divide: UDIV, UDIVcc, SDIV, SDIVcc.
- ☐ Tagged operations that can trap: TADDccTV, TSUBccTV.
- ☐ Load or store FP status registers: LDFSR, STRSR, STDFQ.
- ☐ Branch on floating-point condition code: FBfcc.

**6.6.2.14 Split Before Control Register Read After Previous SetCC**

Integer condition codes are part of the processor status register (PSR). To prevent them from being read while they are being modified, the processor forces an extra cycle between RDPSR instructions and a previous arithmetic operation that may have changed the condition codes.



**6.6.2.15 Split Before MULSCC Unless First One or Two Instructions**

The SSP executes MULSCC as a single instruction group, or it executes two MULSCC instructions as a group. MULSCC is never grouped with any other instructions.

**6.6.2.16 Split Before Extended Arithmetic From CC Set In Current Group**

The SSP cannot use condition codes calculated in the E0 pipeline stage as input to extended precision arithmetic in the same group. This rule inserts a split between the condition code computation and its use in extended arithmetic (ADDX ADDXcc SUBX SUBXcc).

**6.6.2.17 Split Before Delay Group CTI Unless First**

If the group is a delay group, this rule splits the group before the second instruction. This rule prevents additional branches, other than true branch couples, from being executed in the delay group of a branch.

**6.6.2.18 Split Before CTI In JMPL Delay Unless RETT**

This rule allows an optimization of JMPL/RETT pairs and simplifies JMPL couples. A full cycle is required in the pipeline between a JMPL and any other CTI except RETT.

# Instructions

---

---

This chapter describes the operation of SuperSPARC instructions that require a more detailed description than is already provided in *The SPARC Architecture Manual*.

Topic	Page
7.1 Integer Multiply (IMUL) .....	7-2
7.2 Integer Divide (IDIV) .....	7-3
7.3 Write PSR (WRPSR) .....	7-4
7.4 Flush (IFLUSH) .....	7-5
7.5 Store Barrier (STBAR) .....	7-6
7.6 Signal User Emulation Request (SIGM) .....	7-7

## **7.1 Integer Multiply (IMUL)**

SuperSPARC implements integer multiply in all its forms (SMUL, SMUL<sub>cc</sub>, UMUL, and UMUL<sub>cc</sub>), conforming to the SPARC architecture specification. It is mentioned here only because the instruction is new to the Version 8 SPARC architecture.

Integer multiply is implemented in the floating-point unit (FPU) of the processor. Normally, these operations will wait until the completion of any pending floating-point operations (FPOps) before execution (indicated by FP Queue (FQ) empty). If the FPU is in exception mode or exception-pending mode (see Subsection 11.1.9), integer multiply operations will proceed without waiting for the FQ to empty. Integer multiply will not cause any deferred floating-point exceptions to be signalled.

## **7.2 Integer Divide (IDIV)**

For most numeric conditions, integer divide instructions (SDIV, SDIV<sub>cc</sub>, UDIV, UDIV<sub>cc</sub>) work according to the SPARC architecture specification. Due to limitations in the hardware, certain numeric cases cannot be completed. In these cases, SuperSPARC will signal an illegal\_instruction trap (trap type 0x02). This occurs when the 64-bit value comprised of {Y,rs1} has numerically significant bits beyond bit 51. In effect, SuperSPARC only implements a 52-bit by 32-bit integer divide, compared to the 64-bit by 32-bit specification. This holds true for both positive and negative numbers when the operation is a signed divide.

System software is expected to emulate these integer divide operations when required.

Integer divide is implemented in the FPU of the processor. These operations will normally wait until the completion of any pending FPops before execution. If the FPU is in exception mode or exception-pending mode (see Subsection 11.1.9), integer divide operations will proceed without waiting for the FQ to empty. Integer divide will not cause any deferred floating-point exceptions to be signalled.

### 7.3 Write PSR (WRPSR)

SuperSPARC implements the write PSR instruction according to *The SPARC Architecture Manual*, with one qualification. Since SuperSPARC provides no coprocessor port, system software is prevented from trying to enable coprocessor operations.

If a WRPSR instruction attempts to set the PSR.EC bit, an `illegal_instruction` trap will be generated immediately. Since the EC bit can never be set, all coprocessor instructions will generate `cp_disabled` traps (trap type 0x24).

The timing requirements of PSR write operations match *The SPARC Architecture Manual* exactly. A three-instruction (not cycle) delay is required between changing any PSR fields and using the contents. Note also that, when a JMWPL/RETT instruction pair is seen, SuperSPARC will use the PSR.PS (previous supervisor) bit to check protections for the instruction fetch of the JMWPL's target.

The PSR.IMPL (implementation number) field of the PSR is 0x40 and cannot be changed.

## 7.4 Flush (IFLUSH)

SuperSPARC implements FLUSH slightly differently from how *The SPARC Architecture Manual* suggests. No cached information is explicitly flushed by the instruction. The cache-consistency mechanisms are used to ensure that caches always have correct data (both instruction and data caches). The FLUSH operations simply cause an exact synchronization of all pending activity.

When a FLUSH instruction is executed, it will cause SuperSPARC's store buffer to be drained. This causes all pending bus activity to complete. Any cache-coherency transactions (for instance, invalidation of instruction cache entries) will occur as the store buffer clears. The internal processor pipeline and instruction buffer will also be cleared. When the pipeline resumes execution after the FLUSH, it will fetch data from the instruction cache, which is guaranteed to be up to date with respect to all prior processor activity.

---

**Note:**

The FLUSH instruction affects only a single processor. No other processors in a system will do a flush operation unless explicitly requested to do so (by their own FLUSH). This does not matter for most application programs. It is significant, however, to such programs as dynamic linkers. These programs must be written carefully to ensure that all processors see a consistent view of program memory under all circumstances. This may require modifying memory in a certain order and/or writing intermediate values to guard against temporary inconsistencies. Note that cache coherence is maintained on double words.

---

## 7.5 Store Barrier (STBAR)

The STBAR instruction forces all store operations before it to complete before any store operations after it are performed. STBAR is needed only in partial store ordering (PSO) mode. SuperSPARC implements this functionality as described in *The SPARC Architecture Manual*.

The instruction is implemented in the "RDASR" reserved instruction space. The opcode is equivalent to RDASR 0x0f, %g0; the exact encoding is 0x8143c000.

The STBAR instruction sets the SBTAGS.SP in the last allocated (valid) entry. If no entry is allocated, the STBAR instruction is remembered in the bus unit arbitration logic. When the bit is set, the bus unit waits for the PEND\_ input to become inactive before issuing any stores after the one with the bit set. Without the STBAR, the SBTAGS.SP bit will not be set, and the b\_unit will continue issuing stores as fast as the external cache controller can buffer them. This would allow stores to be performed out of order in the system.

All this happens only in PSO mode. In TSO mode, the bus unit waits for PEND at all times.

See Chapter 8 and especially Section 8.7 for more information.

## **7.6 Signal User Emulation Request**

SuperSPARC implements a special SuperSPARC-specific instruction called Signal Emulation (SIGM), that is used in conjunction with the on-chip JTAG-based emulation facilities provided by SuperSPARC.

The instruction is implemented in the "RDASR" implementation-dependent extended opcode space defined by the SPARC architecture. The opcode is equivalent to RDASR 0x1f, %g0, which encodes to 0x8147c000.

The operation of this instruction is dependent on the state of the JTAG controlled MCMD register. If the MCMD.INITM bit is cleared, the SIGM instruction will be executed as a NOP. If the MCMD.INITM bit is set, execution of SIGM will cause immediate entry into emulation mode. (See Section 22.1 for details.)

The MCMD.INITM is always initialized to zero at JTAG Tap controller reset. In order for the SIGM instruction to cause entry into emulation mode, the bit must be explicitly set by a remote emulation processor using JTAG scan. The MCMD.INITM bit cannot be set except through JTAG scan.





# Memory Model

---

This chapter describes the programmer's view of memory in a SuperSPARC-based system. The exact view of memory is highly dependent on system implementation; some of the details below may not apply to certain system environments.

Topic	Page
8.1 Three Models of Memory: SO, TSO, PSO .....	8-2
8.2 Atomic Operations: SWAPs and LDSTUB .....	8-4
8.3 Load and Store Alternates .....	8-6
8.4 Non-Cacheable Loads and Stores .....	8-7
8.5 Page Table Memory Operations .....	8-8
8.6 Prefetch Exception Handling .....	8-10
8.7 Memory Model Support .....	8-11

## **8.1 Three Models of Memory: SO, TSO, PSO**

The SPARC Architecture defines three views of memory: Strong Ordering (SO), Total Store Ordering (TSO), and Partial Store Ordering (PSO). Refer to *The SPARC Architecture Manual* for a comprehensive discussion of the models.

The memory operation of a system depends not only on the processor but also on:

- ☐ Bus or Interconnect,
- ☐ Caches (second, third, and greater levels), and
- ☐ Memory Organization (Banking and Interleaving).

Following is how the SuperSPARC processor supports the three memory models.

### **8.1.1 Strong Ordering (SO)**

Strong ordering allows for maximum software compatibility. Of the three models, SO has the lowest system performance and may have the highest system complexity. In this memory model, all transactions are seen in the single global order in which they were issued by all processors, caches, and memories in the system. It allows for a very simple and intuitive programmer's model.

If the system supports strong ordering, SuperSPARC can implement strong ordering by disabling the internal store buffer (set MCNTL.SB = 0). This will cause decreased performance in most system environments, particularly when using SuperSPARC with the MultiCache Controller (MXCC), since all store operations write through to external caches.

### **8.1.2 Total Store Ordering (TSO)**

Total store ordering is similar to strong ordering, and there is a single global order across all processors of store operations. Furthermore, each processor's store operations always occur in program order. In general, there is a global order for loads. This memory model allows most multi-threaded applications to operate with good performance.

TSO is the normal memory model of SuperSPARC-based systems. TSO is enabled by setting the MCNTL.SB (enabling the store buffer) and clearing MCNTL.PSO (disabling partial store ordering).

### **8.1.3 Partial Store Ordering (PSO)**

Partial store ordering is the highest performing of the memory models. It releases stores from the implied ordering of the program, and requires software to explicitly mark where store ordering is needed. This model requires the most careful use of memory by applications.

The STBAR (store barrier) instruction will guarantee the order of store operations before and after it. All stores issued before a STBAR will complete before any store instruction issued after. To achieve the equivalent of the TSO model, an STBAR might need to be inserted prior to every store operation.

PSO is enabled by setting both the MCNTL.PSO and MCNTL.SB bits to 1. SuperSPARC implements PSO mode in cooperation with external cache and memory controllers. The PENDING signal is sampled in order to determine the completion of store transactions by the system. See Section 8.7.

## **8.2 Atomic Operations: SWAPs and LDSTUB**

SWAP and LDSTUB instructions are used to implement semaphores and other atomic operations in memory.

### **8.2.1 Atomic Operations in the Store Buffer**

Atomic operations force the store buffer to copy out its contents to main memory (store buffer copy-out).

If an exception occurs during the store buffer copy-out caused by an atomic operation, the operation is not completed. Instead, a `data_store_error` is taken, which immediately disables the store buffer. The atomic operation may be restarted when the CPU returns from the store buffer exception handler (see Subsection 10.6.5).

If the atomic operation itself encounters an exception on either the write or read access, a `data_access_exception` will be reported. SuperSPARC guarantees that the destination register will not be updated. The system is responsible for ensuring that the destination memory location, as in any store exception, is not modified.

### **8.2.2 Atomic Operations for SuperSPARC Used With the MultiCache Controller**

Atomic operations have traditionally been implemented as a locked sequence of loads and stores (Read-Modify-Write). In order to support higher-performance packet-switched buses, SuperSPARC implements a true swap operation—it supplies the new data to be written along with the request for the current memory data. This is done to ensure that SuperSPARC receives the current value of the old data. The system or external cache logic must be capable of accepting this transaction. This is made possible by the definition of SPARC's atomic operators—the data to be written is independent of what is read.

### **8.2.3 Atomic Operations for SuperSPARC Directly on MBus**

Since the SuperSPARC processor connected directly to MBus operates with a copy-back, write-allocate cache protocol, the operation of atomic transactions is simpler than when the processor is used with MXCC.

For cacheable references in MBus mode, no special bus operations are done for atomic transactions. They are implemented as a simple sequence of reads and writes. SuperSPARC will read and acquire ownership of the data being referenced, and all operations will occur within the internal cache. If the data is shared, a Coherent Invalidate (CI) bus transaction will be issued to acquire ownership.

Non-cacheable references for SuperSPARC connected directly to MBus operate more traditionally. They will appear on the bus as a locked read-write sequence. The LOCK bit within the MBus address field will be set, and bus arbitration will not be released between the read and write.

### **8.2.4 Alternate Space Atomics**

Swap alternates to ASI locations other than 0x08-0x0b and 0x20-0x2f will cause `data_access_exceptions`. Alternate atomic operations to ASIs 0x08-0x0b and 0x20-0x2f are handled the same as ordinary atomic operations.

### 8.3 Load and Store Alternates

Loads and stores to alternate address spaces are generally performed for low-level control of SuperSPARC and the external system. Appendix B contains a summary list of valid ASIs.

Nearly all ASI operations (both LDA and STA) cause a store buffer copy-out before starting execution. Exceptions are listed in Table 8-1.

*Table 8-1. ASI Operations That do NOT Cause Store Buffer Copy-Out*

ASI	Operation
0x20 – 0x2f	STA
0x40 – 0x4c	LDA/STA
0x08, 0x09, 0x0a, 0x0b	STA
0x08, 0x09, 0x0a, 0x0b	Cacheable LDA

For example, context register writes cause the store buffer to copy-out, preventing the store buffer from containing operations that belong to contexts other than the current one. This allows data\_store\_exceptions to be associated with the faulting process more easily.

If an exception occurs during the store buffer copy-out caused by an LDA/STA, the operation is not completed. A data\_store\_exception is taken, which immediately disables the store buffer. The STA operation may be restarted when the CPU returns from the store buffer trap handler.

Alternate space transactions through ASIs 0xa and 0xb are treated as normal load and store operations. ASIs 0x8 and 0x9 are treated as instruction accesses. These accesses are translated by the MMU and may trigger a table walk. If the table walk encounters an invalid or reserved PTE or PTP, a trap will be taken.

Transactions through the pass-through ASI space (0x20-0x2f) are treated as normal loads and stores, except that they are not translated by the MMU (see Section 9.9). For these operations, cacheability is determined by the MCNTL.AC (alternate cacheable) bit.

## **8.4 Non-Cacheable Loads and Stores**

The cacheability of memory references is determined as described in Subsections 10.2.2 and 10.4.3.

Non-cacheable loads cause the contents of the store buffer to be copied out to memory before proceeding. This is done to ensure proper memory ordering of accesses to I/O space.

If an exception occurs on the store buffer copy-out caused by a non-cacheable load, the load operation is not completed. A `data_store_exception` is taken, which immediately disables the store buffer. The load operation may be restarted when the CPU returns from the store buffer exception handler.

Unlike loads, non-cacheable stores do not normally force the store buffer to copy out. They are simply placed in the store buffer, if enabled, like cacheable stores.



## 8.5 Page Table Memory Operations

The Memory Management Unit (MMU) page tables in system memory should be accessed cautiously because there may be conflicts between software and hardware accesses. This section describes how to access the page tables.

### 8.5.1 Hardware Use of Page Tables

The MMU hardware autonomously reads the page tables to perform translations and modifies them to keep Referenced bits (R) and Modified bits (M) up to date.

When the SuperSPARC MMU table walk hardware accesses the page tables, it does so in a way that guarantees consistency between processors in a multiprocessor system. Table walks are not performed under locks, and hardware and software can interfere with each other if accesses do not follow the algorithm laid out in Subsection 8.5.3.

Accesses by the MMU hardware to update the referenced and modified (R&M) bits in a page table entry are made using standard write or swap operations, and without bus locking. See Section 9.4 for more detailed information on R&M updates.

### 8.5.2 System Software Use of Page Tables

System software accesses page tables to check statistics and remap physical memory.

When only system software accesses page tables in memory, consistency is guaranteed by the normal cache consistency mechanisms, as well as by such standard software access controls as semaphores. Unfortunately, SuperSPARC's table walk hardware has no way to recognize these software-locking conventions. Subsection 8.5.3 describes how software can access the page tables without introducing inconsistencies.

### 8.5.3 Hardware/Software Page Table Consistency

Since both hardware and software accesses to the page tables may be in progress simultaneously, some algorithm must be used to guarantee that these two sources (as well as multiple instances of both of them due to multiple processors) will not interfere with each other.

Inconsistency can occur in several ways. The usual way is when software changes (e.g., invalidates, etc.) a page mapping and the table walk hardware has already read the table entry into the TLB. In this situation, it is the responsibility of the system software to perform an MMU flush (or Demap) operation to force the page table to be re-read by the MMU. See Subsection 9.8.2. This is sufficient in a uniprocessor environment. In a multiprocessor environment, the flush operation must be done on every processor in the system.

**Note:**

Software must guarantee that only a single Demap operation is in progress at any one time across the entire system. Inconsistent operation will result if two Demaps are received by a processor at any one time (including internal Demap requests).

In XBus systems the local flush operation is automatically broadcast to all processors. There is no need to interrupt remote processors to issue local flush transactions. In MBus systems all processors must be interrupted and told to do their own local flush operation. The broadcast Demap capability of XBus is recommended for large, higher-performance multiprocessor systems.

A more difficult problem arises when the MMU hardware must rewrite a page table entry to set or clear the R or M bits. The hardware must be prevented from overwriting a modification that system software has just completed.

A general algorithm that prevents inconsistency in both situations is presented in Example 8-1. The exact implementation of this code is system-dependent. The implementation of lock and unlock operations is memory model dependent; see *The SPARC Architecture Manual* for sequences matching the three memory models.

**Example 8-1. Generalized Safe Page Table Update Algorithm**

```

/* Acquire exclusive page table access */
Lock
/*Any R+M updates will accumulate here*/
RM_Accum = 0
/*Set PTE to zero temporarily */
Loop: Reg = 0
/*Write 0, read back current PTE */
Swap(TargetPTE,Reg)
/*Flush ALL reference to this PTE in system*/
FlushAllMMUs(TargetPTE)
/* Catch any late R+M Changes */
RM_Accum = RM_Accum OR Reg
/* Continue until it's really zero */
if (TargetPTE <> 0) go to Loop
/* Safe to write new value */
TargetPTE = NewPTE
/* Release lock on page table access */
Unlock

```

Operationally, the algorithm may take several iterations to complete. The *FlushAllMMUs* operator is a system-dependent mechanism to execute a flush operation on all MMUs in the system. System software should use the *RM\_Accum* value as the final value that was in the PTE entry before modification. This will guarantee that no page table status information is lost.

## **8.6 Prefetch Exception Handling**

For most cache misses on load or instruction fetch operations, SuperSPARC performs a block read. Block reads will load four double-words from memory into the cache in a burst transaction. SuperSPARC will always request the word that it needs as the first word of the burst, then read the rest of the four double words addressed modulo four. A bus error may be encountered on any one of the double-word transfers.

Demand fetches immediately return data that is required by the processor. If a bus error occurs on a demand fetch, an exception is reported to the pipeline, causing an `instruction_access_exception` to occur.

Non-demand fetches are the remaining words in the burst, also called pre-fetches. If a bus error occurs on prefetch data, it will not be reported to the pipeline. In this case, the entire cache line referenced by this transaction will be invalidated. The demand fetch will have been satisfied, but none of the additional prefetch data is in the processor. If that data is required by the processor in the future, it will be fetched again, as a demand fetch. If the error at that location persists, it will then be reported to the pipeline as an `instruction_access_exception`.

Information about errors of this sort may be accumulated outside the processor for repair or gathering statistics. If desired, external controllers may raise an interrupt to inform system software of the existence of these errors. System software may initiate attempts to eliminate the error at this point before the data is truly required (demapping pages, etc.).

## 8.7 Memory Model Support (PEND)

The operation of SuperSPARC's store buffer and the way in which SuperSPARC uses **PEND** is dependent on the Memory Model selected.

When the store buffer is on (MCNTL.SB = 1), SuperSPARC may be operated in TSO or PSO mode. When in TSO mode, SuperSPARC always waits for **PEND** to go high before issuing another store to either VBus or MBus.

When in PSO mode, SuperSPARC waits for **PEND** to go high before issuing a transaction according to Table 8-2. Note that for some transactions, SuperSPARC waits only if there is a prior store barrier (STBAR) instruction. In PSO, STBAR becomes a barrier in the store buffer so that all stores issued before STBAR are completed before any stores after STBAR are sent from the store buffer to the bus.

Table 8-2. VBus Transactions That Wait On **PEND**

Transaction Type	VBus		MBus	
	PSO	TSO	PSO	TSO
Write-through Store	Only if prior STBAR	Wait	N/A	N/A
Non-cacheable Store	Only if prior STBAR	Wait	Only if prior STBAR	Wait
Swaps	Only if prior STBAR	Wait	N/A	N/A
Non-cacheable Swaps	Only if prior STBAR	Wait	Only if prior STBAR	Wait
Control Space (CSA)	Wait	Wait	N/A	N/A
Demaps	Wait	Wait	N/A	N/A
Internal ASIs	Wait	Wait	Wait	Wait
Copy-back writes	N/A	N/A	Only if prior STBAR	Wait

In the above table the following notation is used:

**Wait** Always wait for **PEND** to go high before beginning the transaction.

**Only if prior STBAR** Wait for **PEND** to go high if a previously issued STBAR is in the store buffer.

Note that internal ASI operations (both loads and stores) will wait for deassertion of the **PEND** signal. There is one exception to this, ASI 0x4c (the "Action on event" register), which does not depend on **PEND**.

When SuperSPARC is used with the MXCC, none of the above changes. MXCC controls SuperSPARC's **PEND** pin. Since the external cache is copy-back, cacheable stores do not generate immediate stores to the bus. When a block is replaced on an external cache miss, the new data is read before any dirty data in the replaced block is stored to the system bus. If the system bus is MBus, stores always occur in order because MBus is busy until the store completes. The XBus interface provides a packet interface, which can allow multiple pending stores to be issued while waiting for acknowledgements to complete. MXCC asserts **PEND (= L)** to SuperSPARC whenever any store has been sent to the system and has not been acknowledged. MXCC will issue multiple Non-Cacheable (NC) stores only if they are to the same page (presumably serialized by the device receiving the packets). MXCC will always force SuperSPARC to wait until the previous operation was completed if the **CCCR.MC** bit (multiple command enable) is clear, thus greatly limiting the opportunities for out-of-order execution by the system.

In summary, while SuperSPARC supports systems that perform stores out of order, SuperSPARC always issues stores in order, and the presence and extent of out-of-order stores is controlled by the system. The system must use the **PEND** pin to signal to SuperSPARC that there are outstanding stores. SuperSPARC uses the **PEND** pin to ensure that the programmer's intentions in the current memory model are obeyed. When the MultiCache Controller is used, it controls the **PEND** pin. MXCC does not reorder stores, can only issue multiple stores in XBus mode, and generally does not issue NC stores out of order.

# MMU Operation

---

---

The SuperSPARC processor (SSP) implements a 64-entry fully-associative Memory Management Unit (MMU) compatible with the SPARC Reference MMU Specification.

The MMU translates 32-bit virtual addresses into 36-bit physical addresses. The mapping is done in units of 4K-byte page, 256K-byte segment, 16M-byte region, or 4G-byte context.

Topic	Page
9.1 Memory Management Unit Fundamentals .....	9-2
9.2 Address Translation .....	9-3
9.3 Large Linear Mappings .....	9-6
9.4 MMU-Referenced and Modified Bits .....	9-8
9.5 TLB Replacement Policy .....	9-9
9.6 Root Pointer and PTP Level2 Cache .....	9-10
9.7 TLB Hit Criteria .....	9-11
9.8 MMU Probe and Demap .....	9-12
9.9 MMU Transparent Mode .....	9-18
9.10 Address Translation Modes .....	9-19
9.11 No-Fault Operation .....	9-20
9.12 MMU Registers (ASI=0x04) .....	9-22
9.13 MMU Translation Look-Aside Buffer (TLB) .....	9-36

## 9.1 Memory Management Unit Fundamentals

The MMU serves to perform a number of important functions in sophisticated computer systems:

☐ Addressing Contexts

Addresses are unique to hardware contexts, corresponding roughly to software processes.

☐ Relocation

Processor-generated addresses are translated to main memory addresses. The translations are relocatable in small blocks called pages.

☐ Controlled Sharing

Portions of the address spaces for pairs of contexts can be set up to be shared between them. The unit of sharing is a page.

☐ Protection

Access can be confined to addressable memory. Each context can be individually allowed or disallowed read, write, and execute access to pages.

☐ Virtualization

A software handler can process translation failures by locating the missing page on backing store, allocating main memory for it, copying it from backing store into main memory, setting up a translation for it to the newly allocated main memory, and retrying the access.

☐ Supervisory Software Support

Special modes and protection modes support supervisory software access to other contexts.

SuperSPARC contains a sophisticated MMU based on the SPARC Reference MMU in *The SPARC Architecture Manual*. This MMU performs translations by stepping through a tree-structured translation table in main memory called the page tables. When a translation is found, it is cached in a Translation Lookaside Buffer (TLB) in the MMU so that the memory accesses to the page tables can be avoided in subsequent accesses.

This chapter describes the page tables and the operation of the MMU and the TLB.

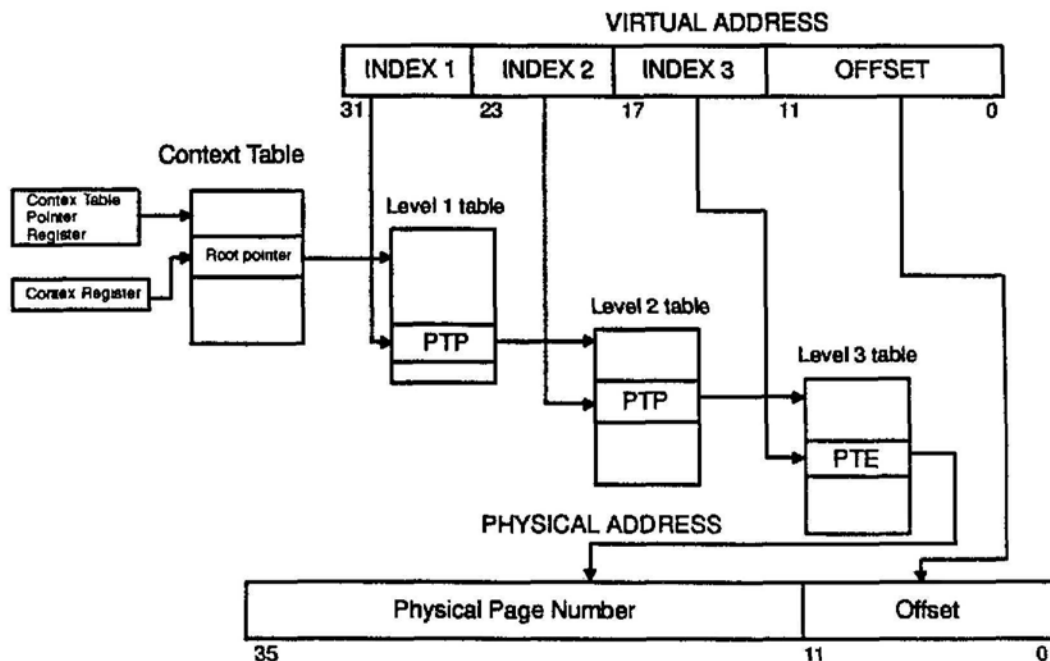
## 9.2 Address Translation

This section briefly describes the software view of memory mapping using the SuperSPARC MMU. For background, refer to the SPARC Reference MMU Specification in *The SPARC Architecture Manual*.

Physical addresses are composed of an offset within a page and a Physical Page Number (PPN), while virtual addresses are composed of an offset within a page and a Virtual Page Number (VPN).

The MMU translates 32-bit virtual addresses and 16-bit context numbers into 36-bit physical addresses by accessing up to four levels of page tables in memory. Normally, this translation is cached in the on-chip 64-entry TLB. When the translation entry is missing from the TLB, the MMU table walk hardware automatically retrieves the translation from the page tables in memory. Figure 9-1 describes the full structure of these page tables.

Figure 9-1. Address Translation Utilizing Four Levels of Page Tables



Each virtual address space is identified by a context number that is kept in the context register. Virtual addresses kept in the TLB are tagged with a 16-bit context number. The effective size of the context register is variable between 10 and 16 bits. This is so that page tables can be smaller in systems with less memory.



The page tables can contain page table pointers (PTP) or page table entries (PTE). A PTE is distinguished from a PTP by the two low-order bits of the table entry (see Figure 9-2). A PTP contains the physical address of the next page table level, while a PTE contains the physical address of the page with its access rights.

### 9.2.1 Page Table Entry

Figure 9-2. Page Table Entry



- PPN** Physical Page Number. This is the high-order 24 bits of the 36-bit physical address. If the PTE maps a 256K-byte segment, 16M-byte region, or 4G-byte context, the lower 6, 12, or 20 bits, respectively, of the PPN are ignored.
- C** Cacheable. If this bit is set to 1, the page is cacheable in the SuperSPARC internal (and external) caches. If it is zero, the page is not cacheable. SuperSPARC asserts the CCHBL pin for transactions involving virtual addresses with the C bit set.
- M** Modified. When a page is accessed for writing and the modified bit is not set, the MMU sets the modified bit (M) in both the TLB and the main memory page table entry.
- R** Referenced. This bit is set to 1 by the hardware when the page is accessed (on a read or a write) and the PTE is missing from the TLB. As with M, both the TLB and the memory copies are set.
- ACC** Access Permissions. This bit field is encoded as shown in Table 9-1. The MMU checks to determine whether an access is authorized according to the mode in which the instruction is executed (user or supervisor) and the type of access (instruction or data reference). If there is a violation of the permissions, a data or instruction access exception is signalled. For more information on access permissions, see MFSR.AT in Subsection 9.12.3.7.

Table 9-1. Access Permission Codes

	ACC	0	1	2	3	4	5	6	7
User	read	✓	✓	✓	✓		✓		
	write		✓		✓				
	execute			✓	✓	✓			
Supervisor	read	✓	✓	✓	✓		✓	✓	✓
	write		✓		✓		✓		✓
	execute			✓	✓	✓		✓	✓

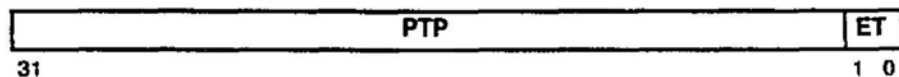
**ET** Entry Type. This field is used to distinguish a PTE from a PTP and to indicate whether a table entry is valid. The encoding is as shown in Table 9-2; a valid PTE always has a ET equal to 2.

Table 9-2. Entry Type Encoding

ET	Entry Type
0	Invalid
1	Page Table Pointer
2	Page Table Entry
3	Reserved

### 9.2.2 Page Table Pointer

Figure 9-3. Page Table Pointer



**PTP** Page Table Pointer. Bits 2 through 31 contain the 30-bit page table pointer. This is the physical address of the base of a next-level page table.

**ET** Entry Type. Distinguishes between a PTE and PTP. The ET field is encoded as in Table 9-2. In a PTP ET is always equal to 1.

**Note:**

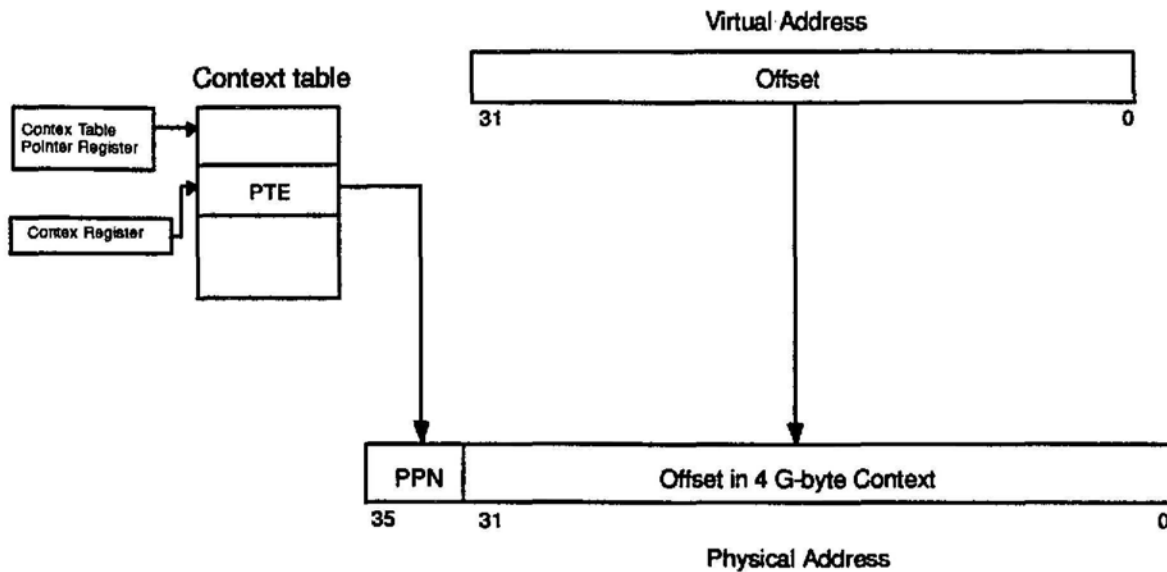
The page tables must be aligned on boundaries equal to their sizes. Low-order bits of the PTP field must be 0. PTPs must point to tables aligned to their natural size.

### 9.3 Large Linear Mappings

The MMU supports mapping sizes larger than the 4K-byte page size. This is done by configuring an entry in the context table, level 1 table, or level 2 table as a PTE (ET=2).

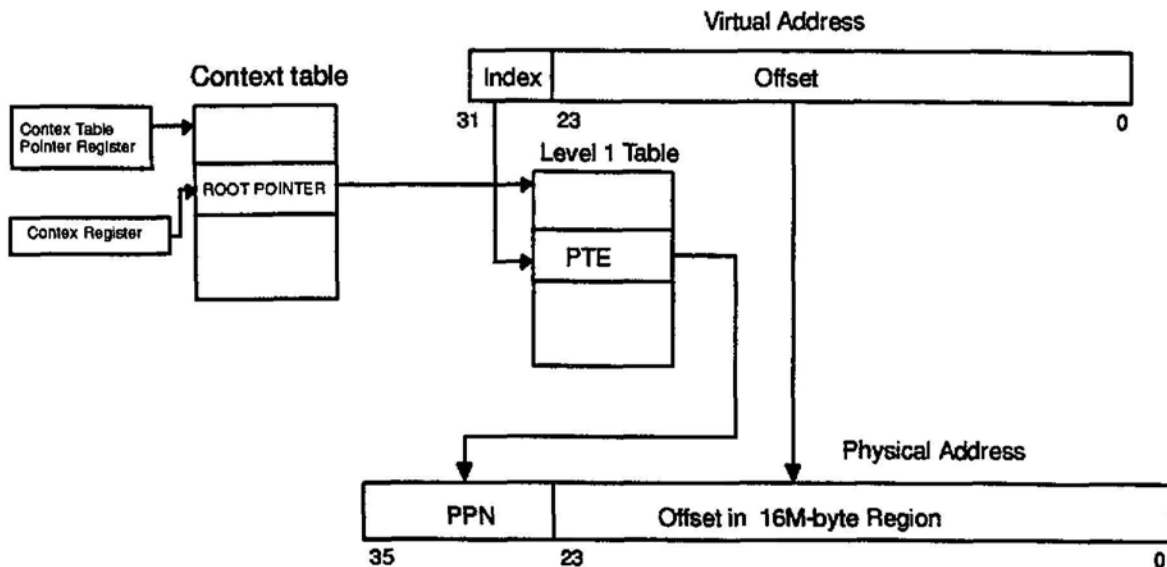
If a PTE is found in the context table, the virtual-to-physical mapping is done as indicated in Figure 9-4 for a 4G-byte context.

Figure 9-4. Address Translation With Maximum Page Size



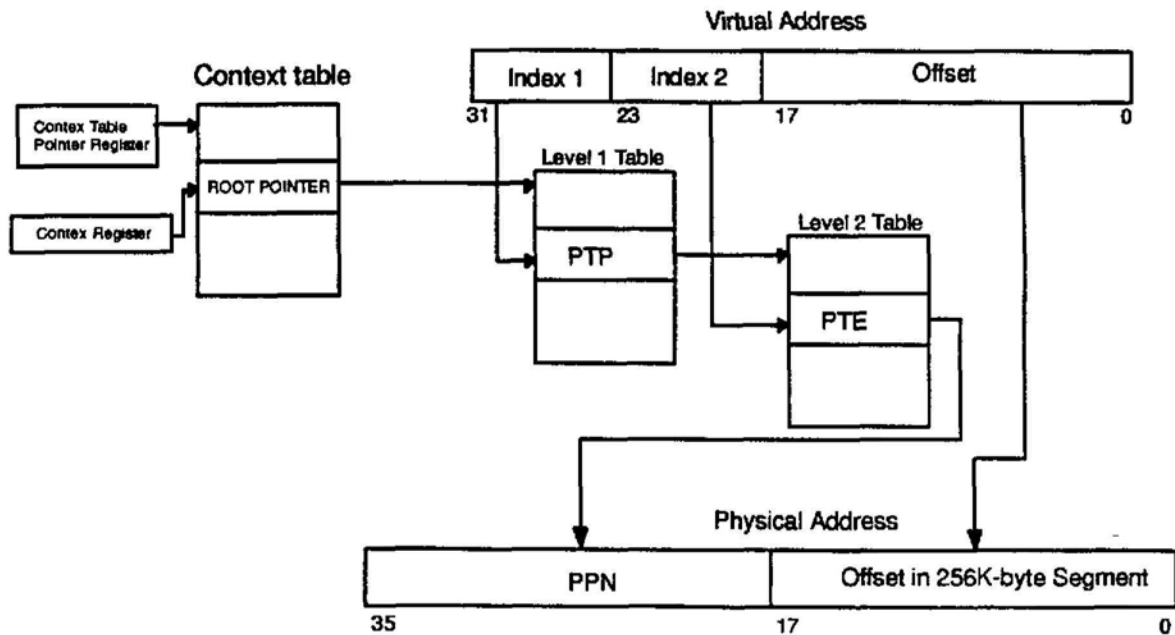
If a PTE is found in a Level 1 table, the virtual-to-physical mapping is done for a 16M-byte region, as shown in Figure 9-5.

Figure 9-5. Address Translation With 16M-byte Page



If a PTE is found in a level 2 table, the virtual-to-physical mapping is done for a 256K-byte segment, as shown in Figure 9-6.

Figure 9-6. Address Translation With 256K-byte Page



The TLB is fully associative, it matches all entries simultaneously. Each entry may be of any of the mapping sizes.

## 9.4 MMU-Referenced and Modified Bits

The referenced bit (R) of a PTE is set to 1 whenever its page is accessed by the MMU during cache miss processing or when an entire probe is initiated. If the Referenced bit (R) is already set, it is not set again.

The Modified bit (M) is checked when the PTE is accessed as a part of the execution of a store instruction. If the M bit is clear in the MMU entry, the M bit in both the TLB and the copy of the PTE in main memory are set to 1.

The R and M bits must be updated in main memory in a manner that guarantees their consistency in a multiprocessor environment. For a discussion of possible algorithms, see Section 8.5.

### *MMU R&M Updates*

Writes to PTEs in main memory by the MMU are required by the SPARC reference MMU architecture to be synchronous; thus they block the execution pipeline. They also force a store buffer copy-out to preserve the sequence of writes (see Section 10.6.) This copy-out is initiated when the memory reference is present at the E0 stage of the execution pipeline.

If an exception occurs on the store buffer copy-out caused by an R&M update, the R&M update operation is not completed. A data store exception is taken, which immediately disables the store buffer. The load, store, or fetch that caused the R&M update will be restarted when the CPU returns from the store buffer trap handler; this in turn will eventually restart the R&M update.

The table walk hardware within SuperSPARC will use a standard write operation when setting both R and M bits in memory. If only the R bit must be set, atomic memory transactions will be used to ensure that another processor is not simultaneously attempting to set both bits. If the processor did not use this protection, it would be possible to overwrite a PTE with both R and M bits set by another updated PTE with only the R bit set. This is prevented by using SWAP transactions to do R bit only updates. Note also that the only combinations that SuperSPARC will ever write back to the PTE are {R=1,M=0} and {R=1,M=1}.

Using SWAP for R-bit updates is essentially the only page table consistency algorithm implemented in hardware. All other cases of page table consistency must be implemented in software as described in Section 8.5.

## 9.5 TLB Replacement Policy

The MMU has 64 TLB entries. When a new PTE entry is brought into the MMU on a TLB miss, it must be stored in the TLB. If one or more entries are invalid, the new translation will be stored in the lowest-numbered invalid entry. When all entries are valid, one of the valid entries must be selected for replacement by the new entry. SuperSPARC's TLB uses a limited-history LRU policy.

Each TLB entry has a used bit associated with it. The used bit is set for any TLB entry that has a TLB hit. When all entries have their used bits set, all used bits (except the last one to be set and those that are locked) are cleared. This is the case when all past history is lost. To select an entry to be replaced when all TLB entries are valid, the lowest-numbered entry for which the used bit is not set will be chosen. Since a TLB hit causes the used bit to be set, this represents the least recently used entry based on the limited history available.

When an entry is invalidated or flushed from the TLB, its corresponding used bit is cleared. When a demap-all operation is done to invalidate all TLB entries, all used bits are cleared. In addition to the used bits, the replacement policy also checks the corresponding lock bit (one per TLB entry). If a lock bit for an entry is set, then, regardless of the used bits, that entry will never be replaced. An invalid entry with its lock bit set can still be replaced, and the newly written entry becomes locked, since the lock bit remains set. If all entries have their lock bits set, no replacement takes place, and the newly brought in PTE is not stored in the TLB. Locking all the entries in the TLB must be avoided, since a translation finishes only after a table-walk operation has completed, thus causing an infinite table-walk loop.

---

### Notes:

Setting all lock bits in the TLB can lead to deadlock and is not recommended.

Allowing the lock bit to be set for invalid entries can lead to inconsistent operation and is therefore not recommended. Since these entries remain locked after a table walk writes a new entry into them, that entry will remain in the TLB. This is true even after a demap operation, which should invalidate it. It is recommended that lock bits be set only in conjunction with explicit writes to that TLB entry by supervisor software.

The lock bits are cleared by hardware reset.

---

## **9.6 Root Pointer and PTP Level2 Cache**

To reduce the time required for each table-walk to access the PTEs, the SuperSPARC MMU caches two special pointers, the root pointer and the PTP from level 2 (PTP2).

The root-pointer for the process in execution is cached. It is invalidated on every context switch, and the first table-walk for the new process will be used to cache in the new root pointer. Caching the root pointer saves the MMU from performing a level of table walk for each TLB miss for that particular context. The root-pointer entry is qualified by a valid bit and implicitly corresponds to the context in the context register. The valid bit for this entry is cleared on context register write, context table pointer write, demap-entire, and demap for a context that matches the context register value.

Caching a level-2 PTP can save memory references during MMU table walks. The SuperSPARC MMU caches one level-2 PTP. The cached level-2 PTP is qualified by a valid bit and implicitly corresponds to the context in the context register. This second-level cached PTP entry is used only for table-walks, R&M bit updates, and probe-entire operations. Other operations still go through the three-level table-walk mechanism. The valid bit for this entry is cleared on context register write, context table pointer write, a table-walk operation not using the cached PTP (in this case, a new entry will be written), demap-entire, demap for a context that matches the context register value, and level-1 and level-2 demaps for which this PTP has a match. The PTP entry is cached on a new table-walk operation and is not cached if the level-2 entry is a PTE.

## 9.7 TLB Hit Criteria

A TLB entry hits for an entry if its validity, virtual address, and access criteria are all met. The criteria for a TLB hit in the MMU are shown in Table 9-3.

Table 9-3. TLB Hit Criteria

Mapping Size	Valid	Vaddr	Access
4K-byte	V = 1	VA[31:12] = VPN	ACC=6 or ACC = 7 or Context = MCTX.CTX
256K-byte	V = 1	VA[31:18] = VPN	ACC=6 or ACC = 7 or Context = MCTX.CTX
16M-byte	V = 1	VA[31:24] = VPN	ACC=6 or ACC = 7 or Context = MCTX.CTX
4G-byte	V = 1		ACC=6 or ACC = 7 or Context = MCTX.CTX

In all cases, the entry must be valid (V=1).

VPN = VA[31:xx] indicates that the corresponding bit fields of the virtual address issued by the processor and the virtual address contained in a TLB Virtual Page Number (VPN) match; see Figure 9-16.

Context = MCTX.CTX denotes that the contents of the context register (see Subsection 9.12.2) and the context field in the MMU entry match. Note that context is not compared for any page classified as a supervisor page by the ACC field being either 6 or 7. In this manner, supervisor pages are present in all contexts simultaneously.

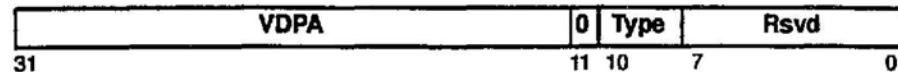


## 9.8 MMU Probe and Demap

The ASI value 0x03 is used to invalidate or probe entries in the MMU. An invalidation of an MMU entry is called a demap.

A probe is done with a load alternate instruction, while a demap is done with a store alternate instruction. The data of the load or store alternate is ignored. The address format for both cases is:

Figure 9–7. Probe and Demap Address Format



The various bit fields have the following meanings:

<b>VDPA</b>	Virtual Demap or Probe Address. See Table 9–5 for the significant virtual address bits.
<b>Type</b>	This field specifies the extent of a demap (mapping size) or the level of the entry probed. See Table 9–4 for the possible levels.
<b>Rsvd</b>	Reserved. These bits are ignored. They should be zeros for compatibility with future versions.

### 9.8.1 MMU Probe

The different types of MMU probes that can be performed and the corresponding returned data are shown in Table 9–4.

Table 9–4. MMU Probe Types

Type	Probe Object
0 (Page 4K-byte)	Level 3 PTE
1 (Segment 256K-byte)	Level 2 and 3 PTE
2 (Region 16M-byte)	Level 1, 2, and 3 PTEs
3 (Context 4G-byte)	Level 0, 1, 2, and 3 PTEs
4 (Entire)	All PTEs
5-7 (Reserved)	

For all the probe operations, the TLB is accessed first. If a translation is found that complies with the matching criteria, the cached PTE is returned. The PTE is returned in the format in Figure 9–2, with PTE.R=1 and PTE.ET=2. The criteria for matching a TLB entry are shown in Table 9–5.

Table 9-5. MMU Probe Operations TLB Matching Criteria

Type	Size	Valid	Access	Vaddr	Level
0	Page – 4K-byte	V = 1	Context = MCTX.CTX	VPN= VDPA[31:12]	level= 3
1	Segment – 256K-byte	V = 1	Context = MCTX.CTX	VPN=VDPA[31:18]	level= 2
2	Region – 16M-byte	V = 1	Context = MCTX.CTX	VPN=VDPA[31:24]	level= 1
3	Context – 4G-byte	V = 1	Context = MCTX.CTX		level= 0
4	Entire	V = 1	Context = MCTX.CTX	(VPN = VDPA[31:12] & level = 3)   (VPN = VDPA[31:18] & level = 2)   (VPN = VDPA[31:24] & level = 1)   level = 0	
5 – 7	Reserved				

In all cases, the entry must be valid (V=1).

VPN = VDPA[31:xx] indicates that the corresponding bit fields of the Virtual Demap or Probe Address issued by the processor and the virtual address contained in a TLB VPN match; see Figure 9-16.

Context = MCTX.CTX denotes that the contents of the context register (see Subsection 9.12.2) and the context field in the MMU entry match.

Level denotes the level at which the PTE is found.

If the PTE is not found in the MMU, a table walk is initiated. In this case, the data returned may not be a PTE. The returned data can be a PTP or, if an error occurs, the value 0. The error cases are slightly different from the ones that may occur during a regular table walk when the MMU is processing a miss. They are detailed below. If an error occurs during a probe table walk, no exception is taken, but the AT field of the MMU fault status register (MFSR.AT) is set to 1 (load from supervisor data space), the MFSR.FT field to 1 (invalid address error) or 4 (translation error), and the MFSR.L field to the table level where the error was detected (see Subsection 9.12.3.7).

- ☐ **Page Probe (4K-byte).** For a page probe, the hardware does a table walk and returns the PTE found in the level 3 table even if this level 3 entry is invalid (PTE.ET=0). A table walk may not complete correctly for any of the following reasons:

- The level 3 entry accessed is not a PTE (PTE.ET=1 or 3);
- An intermediate-level entry is not a PTP (PTE.ET = 1);
- A hardware error occurs.

In these cases, the value 0 is returned, and the MFSR.FT is set to 4, which indicates that a translation error has occurred.

- ☐ **Segment (256K-byte) or Region (16M-byte) Probe.** For a segment or region probe, a PTE (PTE.ET=2) or a PTP (PTE.ET=1) is returned from the level 2 or level 1 table entry, respectively, even if it is invalid (PTE.ET=0). The table walk may not complete for any of the following reasons:

- The entry accessed is reserved (PTE.ET=3);
- An intermediate level is not a PTP (PTE.ET = 1);
- A hardware error occurs.

In these cases, the value 0 is returned, and the MFSR.FT is set to 4, which indicates that a translation error has occurred.

- ☐ **Context Probe (4G-byte).** For a context probe, the level 0 entry accessed is returned if it is a PTE (PTE.ET=2) or a PTP (PTE.ET=1) or is invalid (PTE.ET=0). The value 0 is returned and the MFSR.FT is set to 4 (translation error) if the entry is reserved (PTE.ET=3), unless a hardware error occurs.

- Entire Probe. For an entire probe, the hardware does a regular table walk and returns the valid PTE found in the appropriate level table. A PTE may not be found for any of the following reasons:
  - An invalid (PTE.ET=0) or reserved entry (PTE.ET=3) is accessed in an intermediate level;
  - The level 3 entry accessed is a PTP (PTE.ET=1);
  - A hardware error occurs.

In these cases, the value 0 is returned. If an invalid entry is accessed, the MFSR.FT field is set to 1 (invalid address error); in the other cases, the FT field is set to 4 (translation error).

When an entire probe completes successfully, the PTE accessed is loaded in the TLB, and, if necessary, the R bit is updated. For all the other probe operations, the TLB is left unchanged and the R bit is not updated.

**Note:**

SuperSPARC generates a `data_access_exception` on probe types 0x5-0x7.

## 9.8.2 Demap

The different types of demaps and the objects invalidated in the TLB are shown in Table 9-6.

Table 9-6. Demap Types

Type	Flush Object
0 (Page 4K-byte)	Level 3 PTE
1 (Segment 256K-byte)	Level 2 and 3 PTE
2 (Region 16M-byte)	Level 1, 2, and 3 PTEs
3 (Context 4G-byte)	Level 0, 1, 2, and 3 PTEs
4 (Entire)	All PTEs
5-7 (Reserved)	

The TLB hit criteria for a demap are shown in Table 9-7.

Table 9-7. TLB Demap Hit Criteria

Type	Size	Access	Vaddr	Level
0	Page – 4K-byte	Context = MCTX.CTX or ACC = 6 or ACC = 7	VPN=VDPA[31:12]	level= 3
1	Segment – 256K-byte	Context = MCTX.CTX or ACC = 6 or ACC = 7	VPN=VDPA[31:18]	level= 3 or 2
2	Region – 16M-byte	Context = MCTX.CTX or ACC = 6 or ACC = 7	VPN=VDPA[31:24]	level= 3 or 2 or 1
3	Context – 4G-byte	Context = MCTX.CTX and ACC < 6		
4	Entire	none		

VPN = VDPA[31:xx] indicates that the corresponding bit fields of the Virtual Demap or Probe Address issued by the processor and the virtual address contained in a TLB Virtual Page Number (VPN) match; see Figure 9-16.

Context = MCTX.CTX indicates that the contents of the context register (see 9.12.2) and the context field in the MMU entry match. Note that context is not compared for any page classified as a supervisor page due to the ACC field being either 6 or 7. In this manner, supervisor pages are present in all contexts simultaneously.

Level denotes the level at which the PTE is found.

In a multiprocessor system, distinct processors can hold copies of the same PTE in their MMUs. Therefore, when a portion of a virtual space is demapped, the demap operation should be applied to all MMUs in the system.

Depending on the system configuration, demap operations may be broadcast automatically to all processors or may require explicit software intervention on all processors to demap the required pages. SuperSPARC allows for automatic demapping only when used with the MultiCache Controller (MXCC). MXCC implements system demaps on XBus. When SuperSPARC is used with an MXCC on MBus or when it is used directly on MBus without an MXCC, all processors must be interrupted and requested to flush their own TLBs whenever the page table is modified. See also Section 8.5.

**Note:**

Software must guarantee that only a single demap operation is in progress at any one time across the entire system. Inconsistent operation will result if two demaps are received by a processor at any one time (including internal demap requests).

When an MMU demap has been completed by all processors (either through hardware or software), the following should be true:

- ☐ All memory references to the virtual space(s) mapped by the flushed PTE(s) that were issued before the demap must have been completed.
- ☐ No MMU has a valid copy of the PTE(s).
- ☐ All memory references to the PTE(s) itself (themselves) that were issued before the demap have been completed.

Once the system has reached this state, page tables may safely be modified and applications allowed to continue.

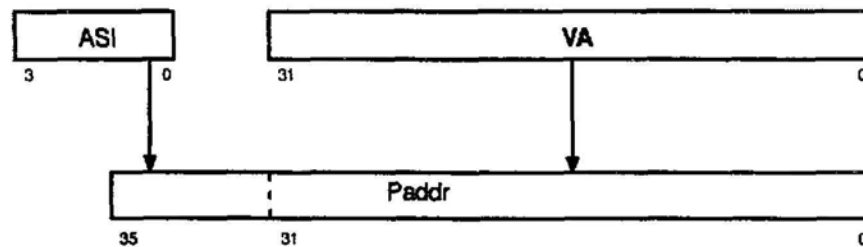
## 9.9 MMU Transparent Mode

The ASI values 0x20-0x2f are used to bypass the MMU for data accesses. The MMU does not translate the physical address through the TLB. The virtual address is translated as follows (illustrated in Figure 9-8):

$ASI[3:0] \rightarrow Paddr[35:32]$ ,  $VA[31:0] \rightarrow Paddr[31:0]$ .

Cacheability of these accesses is determined by MCNTL.AC bit (see Subsection 9.12.1). Since these transactions are completely based on physical memory addresses, they can have no effect on virtual memory components, such as the MMU R&M bits.

Figure 9-8. MMU Transparent Mode Translation



## 9.10 Address Translation Modes

Table 9–8 summarizes the different translation modes used by the Super-SPARC processor.

Table 9–8. Address Translation Modes

Translation Mode	Instruction Fetch (ASI=0x08 or 0x09)	Data Access Mode (ASI=0x0a or 0x0b)
Boot Mode MCNTL_BT=1 MCNTL_EN=X	PA[35:28]=0xff PA[27:0]=VA[27:0]	EN=0 → Disabled Mode EN=1 → Enabled Mode
MMU Disabled MCNTL_BT=0 MCNTL_EN=0	PA[35:32]=0x0 PA[31:0]=VA[31:0]	PA[35:32]=0x0 PA[31:0]=VA[31:0]
MMU Enabled MCNTL_BT=0, MCNTL_EN=1	PA[35:12] from PTE PA[11:0]=VA[11:0]	PA[35:12] from PTE PA[11:0]=VA[11:0]
MMU Transparent	Not Applicable	LDA and STA with ASI=0x20-0x2f PA[35:32]=ASI[3:0], PA[31:0]=VA[31:0]

BT is the Boot Mode bit of the MMU control register.  
 EN is the MMU Enable bit of the MMU control register.  
 PA[bit range] are the bits of the physical address.  
 VA[bit range] are the bits of the virtual address.

The ASIs used above are:

- ☐ 0x08 – User Instruction Space.
- ☐ 0x09 – Supervisor Instruction Space.
- ☐ 0x0a – User Data Space.
- ☐ 0x0b – Supervisor Data Space.
- ☐ 0x20-0x2f – MMU Transparent Mode (see Section 9.9).



## 9.11 No-Fault Operation

The MCNTL.NF bit, when enabled, turns on no-fault operation. In this mode, most exceptions generally reported to the pipeline are disabled. This mode is intended for use by system software during the processing of exceptions and during system diagnostic functions. The NF bit should not be set during user code execution.

Any memory transaction that has an error or an exception while in No-Fault mode does not trap, except for the cases below. If an exception occurs in the case of load transactions, the destination register will be updated with indeterminate information. Should an exception occur in the case of stores, no registers will be updated, and the system designer becomes responsible for ensuring that the system does not modify memory.

When operating with NF set, the success or failure of every memory transaction should be verified by explicitly reading the MFSR fault status register.

In general, all normal memory exceptions are disabled by NF. There are several types of exceptions that are not disabled by NF. The exception types that are not disabled are all considered fatal errors (not recoverable) and will induce error mode. The following exceptions are not disabled by NF:

☐ Internal Error

Errors such as multiple tag matches from the caches are considered fatal internal errors.

☐ Control Space Error

Errors Reading or Writing SuperSPARC ASI registers are considered fatal.

☐ Supervisor Instruction (ASI 0x09)

Supervisor instruction fetch errors cannot be disabled by NF, since the processor has effectively received an error in the instructions it needs to execute. Since there is no other source for instructions, an exception must be generated.

☐ User Instruction (ASI 0x08)

Instruction fetches (explicitly not alternate space read and writes) from ASI 0x08 (User Instruction space) will cause exceptions to be reported. As above, no instructions could be executed without the exception.

☐ Unassigned ASIs

Unassigned ASIs will trap regardless of MCTL.NF.

---

**Note:**

It is the responsibility of system software to ensure that, upon return to user code, the NF bit is never set. Any load operation that receives an exception masked by the NF bit will load indeterminate data to the destination register. Any store that receives an exception masked by NF will have no effect on registers (guaranteed) or memory (system dependent).

---

## 9.12 MMU Registers (ASI=0x04)

Accesses to ASI 0x04 read and write SPARC Reference MMU control registers. These registers are all 32 bits wide, and their addresses are shown in Table 9-9. Attempts to access them with Byte, Half-word, or Double-word operations will result in `data_access_exception`. Virtual address bits [12:8] are used to select individual registers; all other bits are ignored and should be 0. Any access to addresses other than those defined in Table 9-9 causes a `data_access_exception`.

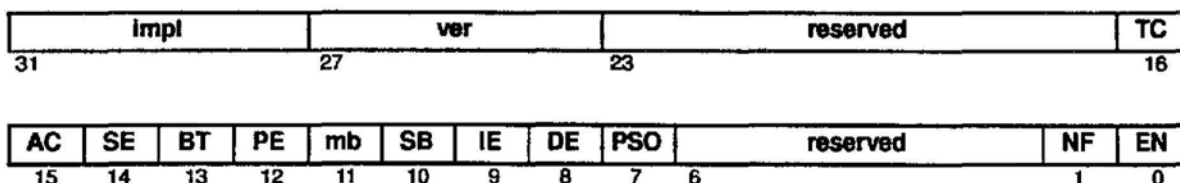
Table 9-9. MMU Control Registers

VA[12:00]	Description
0x0000	Control Register (MCNTL)
0x0100	Context Table Pointer Register (MCTP)
0x0200	Context Register (MCTX)
0x0300	Fault Status Register (MFSR)
0x0400	Fault Address Register (MFAR)
0x1300	Read/Write Fault Status Register
0x1400	Read/Write Fault Address Register
0x1500	Shadow Fault Status Register (MSFSR)

### 9.12.1 MMU Control Register (MCNTL)

The control register contains the general MMU control and status flags. It also contains the control flags for the instruction and data caches. The MMU control register is shown in Figure 9-9.

Figure 9-9. MMU Control Register



**impl** Implementation number of the SuperSPARC chip; this field is hardwired and is read-only. SuperSPARC fixes MCNTL.Impl = 0x0.

**ver** Version number of the SuperSPARC chip; this field is hard-wired and is read-only.

<b>reserved</b>	Reserved. These bits are ignored on write and read as zero.
<b>TC</b>	<p>Table Walk Cacheable Bit. When this bit is set, references from MMU table walks are cached in the SuperSPARC external cache. When this bit is cleared, table walk accesses are not cached in the SuperSPARC external cache. Table walk references are never cached internally. When SuperSPARC is operating directly on MBus (with no MXCC), TC must be deasserted.</p>
<b>AC</b>	<p>Alternate Cacheable bit. This bit indicates whether an access is cacheable in the absence of the C bit in the PTE due to the MMU being disabled or in a mode where the PTEs are not needed for translation. The only exception to this case is the instruction fetches in boot mode, which are always non-cacheable. When this bit is clear, memory accesses for which the physical address is not obtained through a PTE are not cached in the SuperSPARC internal caches or the external cache. If this bit is set, these memory accesses are cached.</p> <p>The cacheability of MMU table walks are controlled by TC and not by this bit.</p>
<b>SE</b>	<p>Snoop Enable. This bit, when set, enables cache snooping on the SuperSPARC bus. This bit must be set to enable the cache-consistency mechanisms. Assertion of this bit does not affect store buffer snooping, which is always enabled if the store buffer is enabled.</p> <p>In an SSP operating directly on the MBus (without MXCC), both the instruction and data caches will snoop regardless of whether they are enabled (IE and DE bits) and without regard to the state of SE. This is necessary for maintaining consistency should any of the caches be disabled after they contain valid data. Therefore, initialization code should contain a flash clear for these caches before enabling snoops at power-on reset. This is to prohibit garbage data from residing in the snooping caches.</p>
<b>BT</b>	<p>Boot Mode. A clear BT bit indicates normal SPARC reference MMU operation. A set BT bit indicates boot mode. Physical addresses for instruction fetches in boot mode are formed by adding the address 0xff000000 to the lower 28 bits of the virtual address. Instruction accesses in boot mode bypass the SuperSPARC internal cache. Data accesses are unaffected by the boot mode.</p>

<b>PE</b>	Parity Enable. When set, even parity is generated and checked by SuperSPARC. When zero, odd parity is generated but not checked.
<b>mb</b>	MBus Mode. This bit, if set, indicates that the SSP is connected directly to the MBus without an MXCC. This bit is read-only and reflects the state of the <code>CCMODE</code> pin.
<b>SB</b>	Store Buffer Enable. This bit enables store buffer operation. When clear, stores do not go to the store buffer and occur synchronously (see Chapter 8). When the SB bit is set, stores are buffered in the store buffer (see Section 10.6) and complete as soon as entered.
<b>IE</b>	Instruction Cache Enable. This bit enables the instruction cache (see Section 10.2).
<b>DE</b>	Data Cache Enable. This bit enables the data cache (see Section 10.4).
<b>PSO</b>	Partial Store Ordering. When set, the memory model is in Partial Store Ordering (PSO) mode. When clear, it is in Total Store Ordering (TSO) mode (see Chapter 8). The SB bit must be set (enabling the store buffer) to get either PSO or TSO mode.
<b>reserved</b>	Reserved. These bits are ignored.
<b>NF</b>	No-Fault Bit. When this bit is asserted, faults that occur for ASIs 0x08, 0x0a, 0x0b, and 0x20-0x2f are ignored (not reported to the processor). The remaining ASIs, including ASI 0x09, SuperSPARC internals, and control space (ASI 0x02), are not affected by this bit. Note that the MFSR is always updated regardless of whether the fault is taken (see Section 9.11).
<b>EN</b>	MMU Enable. This bit enables or disables the operations of the MMU. If the BT bit is set, the EN bit is effective for data cycles only; instruction fetches operate with boot mode translations. When the MMU is disabled, the virtual address is used as the physical address without translation.

On power-on reset, all the control bits in MCNTL are cleared, except for the BT bit, which is set.

### 9.12.2 Context Table Pointer Register (MCTP) and Context Register (MCTX)

The context table pointer register (MCTP) contains a pointer to the context table in physical memory. The context table pointer register is shown in Figure 9-10.

Figure 9–10. Context Table Pointer Register (MCTP)

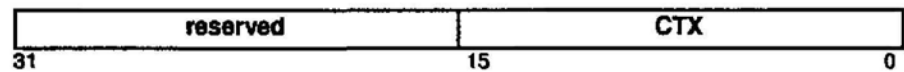


**CTP** Context Table Pointer. The context table pointer contributes up to 24 bits of the root pointer.

**reserved** Reserved. These bits are ignored on writes and read as zero.

The Context Register (MCTX) contains the displacement in the context table to access the root pointer. It defines the current virtual address space. The context register has the structure shown in Figure 9–11.

Figure 9–11. Context Register (MCTX)

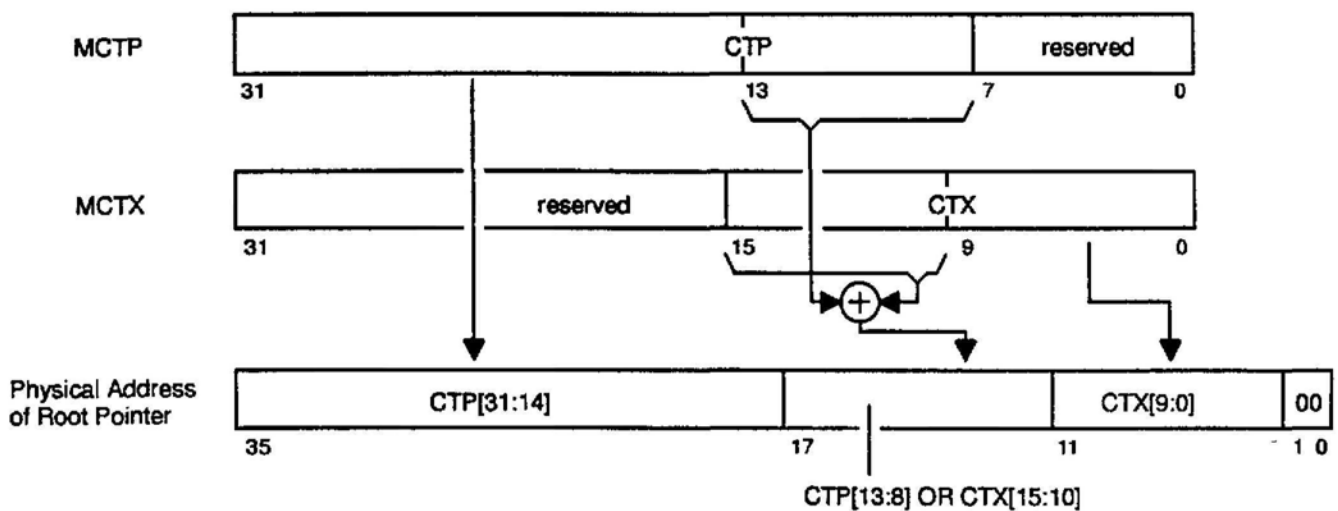


**reserved** Reserved. These bits are ignored on writes and read as zero.

**CTX** Context Number. The context number contributes up to 16 bits in the root pointer.

The context table contains the root page table pointers for all the contexts. The MCTX provides the offset into the context table to retrieve the root page table pointer for that context. The physical address used to retrieve the root page table pointer is formed as shown in Figure 9–12.

Figure 9–12. Root Pointer Physical Address Generation



This address formation gives the kernel the freedom to trade off a number of context bits against alignment restrictions on the context table. Note that, if a context of 16 bits is desired, the context table must be aligned to a 256K-byte boundary.

Table 9-10 gives the alignment requirements for the context table for different widths for the context register.

*Table 9-10. Alignment Requirements*

Context bits	Alignment
$\leq 10$	4K
11	8K
12	16K
13	32K
14	64K
15	128K
16	256K

### 9.12.3 MMU Fault Status Register (MFSR)

The fault status register provides information for SuperSPARC faults associated with the memory system and other internal error sources. The MFSR, along with the reported trap type, are used to distinguish between the various types of errors and faults that can occur.

There are several general types of exceptions: instruction access faults, data access faults, store buffer exceptions, and internal errors.

#### 9.12.3.1 Instruction Access Errors

Instruction access faults are signalled through the `instruction_access_exception` trap. There are several possible sources for the exception. It may be created by normal page faults reported by the MMU. MMU-generated errors are distinguished by the encoding of the MFSR.FT (fault type) field (see Subsection 9.12.3.7), indicating the MMU fault type. The fault may be externally generated for such things as bus timeouts and parity errors. Externally generated faults are indicated by various bits in the MFSR that are tied to equivalent error responses on the external busses. A final source of errors is from internally detected inconsistencies. For example, in the case of a multiple tag match in the instruction cache, SuperSPARC enters error mode and generates a watchdog reset, and the MFSR.EM (error mode) bit (see Subsection 9.12.3.7) will be set.

### 9.12.3.2 Data Access Errors

Data access faults are signalled through the `data_access_exception` trap. As with instruction access errors, page faults, external bus errors, and internal errors may all cause exceptions that are indicated in the various fields of the MFSR register. Additional errors can be caused by erroneous accesses to internal ASI control spaces. These are indicated by the MFSR.CS (control space access error) status bit (see Subsection 9.12.3.7).

### 9.12.3.3 Store Buffer Errors

Store buffer errors are signalled as `data_store_error` traps. These errors are asynchronous exceptions; they are reported after the apparent completion of the instruction that caused them. When this type of error occurs, the MFSR.SB (store buffer error) bit will be asserted. The source of the error is generally a parity, timeout, or similar error on the bus. The erroring store has already been successfully translated by the MMU; otherwise, a `data_access_exception` would have been reported. The operation can be recovered by reading the contents of the store buffer and explicitly completing the transaction using transparent MMU physical address memory references or other means. Depending on the error, software may be able to restore the system state and continue operation. See Section 10.6 for a detailed description of store buffer errors and operation.

### 9.12.3.4 Control Space Access Errors

Illegal ASI operations will cause `data_access_exceptions` and set the MFSR.CS (control space access error) bit.

Any of the following four conditions can cause these errors:

- ☐ References to an invalid ASI.
- ☐ References to a legal ASI, but with an invalid data size.
- ☐ An invalid virtual address field within a valid ASI.
- ☐ Bus error on ASI 0x02 (control space).

The MFSR.CS bit is not asserted under either of the following conditions:

- ☐ Bus error on ASI 0x08, 0x09, 0x0a, 0x0b, 0x20-0x2f (the standard and bypass ASIs).
- ☐ Errors on MMU probes.

The contents of the Fault Address Register (MFAR) is set to the faulting address when MFSR.CS (control space access error) is set.

### 9.12.3.5 Error Mode and Internal Errors

The SPARC Architecture specifies that a processor that takes a precise or new deferred exception with `PSR.ET = 0` will enter error mode.



SuperSPARC will also enter error mode if an internal error occurs. An example of this is detection of multiple tag matches within the instruction and data caches. In these cases, the MFSR.FT field will be set to 6, indicating an internal error.

Whenever SuperSPARC enters error mode, MFSR.EM is set. SuperSPARC exits error mode by taking a watchdog reset. MFSR.EM should be examined by the reset handler to distinguish software-induced error conditions from hardware reset.

---

**Note:**

When internal error occurs and watchdog reset is taken, only MFSR.EM and MFSR.FT are meaningful. The states of the other MFSR bits are not guaranteed.

---

### 9.12.3.6 MFSR Timing and Operation

The MFSR is guaranteed to be valid only after instruction, data, and store buffer exceptions (instruction\_access\_exception, data\_access\_exception, and data\_store\_error, respectively). The contents of the fault status register can be misleading if examined at an arbitrary time. For example, an instruction fetch that is not a demand fetch (not needed immediately for execution) may cause the fault status register to indicate a fault that is never signalled as an actual exception.

Read access to the MFSR is at virtual address 0x00000300 using ASI 0x04. When read from this address, the MFSR is cleared. MFSR can be read and written at virtual address 0x00001300 using ASI 0x04.

The MFSR.SB (store buffer) error bit is sticky. In other words, once it is set, it will not be overwritten by the occurrence of any other exceptions. This is to ensure that store buffer error events are never lost. The SB bit will be cleared on any read of the MFSR; it can also be cleared by writing explicitly to the read/write version of the MFSR.

**Note:**

Translation errors are considered high-priority errors. The occurrence of a translation error cannot be overwritten by any other errors. Even if the exception associated with a translation error is not taken (due to other exceptions, prefetches, branches, etc.), the MFSR.FT bit will continue to indicate the occurrence of a translation error.

This implies that, under certain circumstances, a trap may be incorrectly stated to be a translation error when in fact it may have been due to other causes. System software should be able to recover from this situation by using probe operations to test the validity of translations, and, if correct, retrying the instruction that reported the exception. If the true source of the exception continues to exist, it will be reported again.

Translation errors may be overwritten by other translation errors, in which case the MFSR.OW (overwrite) bit will be set.

General rules for overwriting the MFSR (and MFAR) are presented in Table 9-11. The MFSR.OW bit will always be set if an overwrite condition occurs.

*Table 9-11. MFSR Overwrite Operations*

Pending Error	New Error	OW Status	Action Signalled
Translation Error	Translation Error	Set	Translation Error
Translation Error	Data Access Exception	Unchanged	Data access Exception
Translation Error	Instruction Access Exception	Unchanged	Instruction Access Exception
Data Access Exception	Translation Error	Clear	Translation Error
Data Access Exception	Data Access Exception	Set	Data access Exception
Data Access Exception	Instruction Access Exception	Unchanged	Instruction access Exception
Instruction Access Exception	Translation Error	Clear	Translation Error
Instruction Access Exception	Data Access Exception	Clear	Data Access Exception
Instruction Access Exception	Instruction Access Exception	Set	Instruction Access Exception

In all cases in which simultaneous errors occur, SuperSPARC will choose the highest-priority error and update the status accordingly. The positional priority described above must also be taken into account. The priority order is listed in Table 9-12 (priority 1 is the highest priority).

Table 9–12. Priority Order for Errors

Error	Priority
Translation Error	1
Data Access Exceptions	2
Instruction Access Exceptions	3

## 9.12.3.7 MFSR Register Description

reserved																EM	CS
31																17	16
SB	P	UD	UC	TO	BE	L		AT		FT		FAV	OW			1	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		

<b>reserved</b>	Reserved. These bits are ignored on write and read as zeroes.
<b>EM</b>	Error Mode Reset. When set, this bit indicates that an Error Mode Reset has been taken.
<b>CS</b>	Control Space Access Error. This bit is set to signal errors during ASI accesses. See Subsection 9.12.3.4.
<b>SB</b>	Store Buffer Error. When set, this bit indicates that a store buffer error (data_store_error) has occurred.
<b>P</b>	Parity Error. When set, this bit indicates that the internal parity checkers detected a VBus parity error. Parity errors also set the MFSR.UC bit.
<b>UD</b>	Undefined Error. This bit is set for external bus errors when the external system signals an unfinished error. UD/UC/TO/BE errors are mutually exclusive.
<b>UC</b>	Uncorrectable Error. This bit is set for external bus errors for which an uncorrectable error occurred. Parity errors and ECC errors are also reported as uncorrectable errors. UD/UC/TO/BE errors are mutually exclusive.
<b>TO</b>	Time-Out. When this bit is set, it indicates that a time-out error occurred for an external bus transaction. UD/UC/TO/BE errors are mutually exclusive.
<b>BE</b>	Bus Error. When this bit is set, it indicates that a bus error occurred on a faulting access. This includes invalid bus transactions. UD/UC/TO/BE errors are mutually exclusive.

**Note:**

The bits EM/CS/SB/P/UD/UC/TO/BE in MFSR are defined in the SPARC Architecture as the EBE field (MFSR[17:10]). This EBE field always records the latest bus error cause and in some cases may deviate from the SPARC architecture rule that states:

“A lower priority fault may not overwrite the MFSR status of a higher priority fault.”

If a lower-priority fault occurs after a higher priority fault but before the MFSR has been read, the MFSR should remain unchanged. SuperSPARC complies with this rule for every bit, except for the EBE field.

For example: A data table walk suffers timeout, and MFSR correctly records the status of this translation fault, setting MFSR.[AT,FT,TO] correctly. Then a demand fetch suffers bus error; this is an instruction access fault, which is a lower priority than a translation fault. MFSR.[AT,FT] are correctly unchanged (retaining the status of the translation fault instead of the new instruction access fault), but MFSR.TO is not retained. Instead, the EBE field is updated and MFSR.BE is set.

**L** Level. This field is set to the page table level of the entry that caused the fault. If an external bus error is encountered while fetching a page table entry (either a PTE or PTP), the level field records the page table level for the entry. The field is defined in Table 9-13.

Table 9-13. Field Levels

L	Level
0	Root pointer
1	Level 1 entry
2	Level 2 entry
3	Level 3 entry

**AT** Access Type. This field defines the type of access that caused the fault. See Table 9-14.

Table 9–14. Access Types

AT	Access Type
0	Load from User Data Space
1	Load from Supervisor Data Space
2	Load/Execute from User Instruction Space
3	Load/Execute from Supervisor Instruction Space
4	Store to User Data Space
5	Store to Supervisor Data Space
6	Store to User Instruction Space
7	Store to Supervisor Instruction Space

FT

Fault Type. This field defines the type of the current fault. See Table 9–15.

Table 9–15. Fault Types

FT	Fault Type
0	None
1	Invalid address error
2	Protection error
3	Privilege violation
4	Translation error
5	Access bus error
6	Internal error
7	Reserved

FT = 1. The invalid address error code is set when an invalid PTE or PTP is found while fetching an entry from the page table for a regular table-walk or a probe operation.

FT = 2. The protection error code is set if an access is attempted that is inconsistent with the protection attributes of the corresponding page table entry.

FT = 3. The privilege error code is set when a user program attempts to access a supervisor-only page.

FT = 4. A translation error code is set when an external bus error, reserved PTE, or level-3 PTP is found while fetching an entry from a page table for a regular table-walk or a probe operation. The L field records the page table level at which the error occurred for the above two error codes. The UD, TO, BE, and UC fields record the type of bus error, if any.

FT = 5. An access bus error code is set when an external bus error occurs during memory access.

FT = 6. The internal error code is set when either cache detects an internal inconsistency, such as multiple matches for a particular request. Internal error causes the chip to enter error mode, causing a watchdog reset.

Invalid address errors, protection errors, and privilege violations are a function of the access type and the ACC field of the corresponding PTE (as shown in Table 9-16). The errors are set as in Table 9-17.

Table 9-16. Access Permission Codes

ACC	Permissions	
	User	Supervisor
0	Read Only	Read Only
1	Read/Write	Read/Write
2	Read/Execute	Read/Execute
3	Read/Write/Execute	Read/Write/Execute
4	Execute	Execute
5	Read	Read/Write
6	-	Read/Execute
7	-	Read/Write/Execute

Table 9-17. Access Fault Type (FT Field)

AT	FT Code								
	PTE[V]=0	PTE[V]=1, PTE[ACC]=							
		0	1	2	3	4	5	6	7
0	1	-	-	-	-	2	-	3	3
1	1	-	-	-	-	2	-	-	-
2	1	2	2	-	-	-	2	3	3
3	1	2	2	-	-	-	2	-	-
4	1	2	-	2	-	2	2	3	3
5	1	2	-	2	-	2	-	2	-
6	1	2	2	2	-	2	2	3	3
7	1	2	2	2	-	2	2	2	-

**FAV** Fault Address Valid. This bit is asserted if the contents of the MFAR are valid. The MFAR is not valid for instruction access faults (see Subsection 9.12.4).

**OW** Overwrite. This bit is asserted if the fault status register (MFSR) has been written more than once by faults of the same class since the last time it was read (see Subsection 9.12.3.6).

#### 9.12.4 MMU Fault Address Register

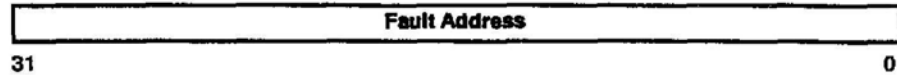
The MFAR records the virtual address of the same faults reported in the MFSR. The MFAR is overwritten according to the policy defined for the MFSR (see Subsection 9.12.3.6). The MFAR is read-only according to the reference MMU specification. For diagnostic purposes, SuperSPARC has additional read/write access to the MFAR using virtual address 0x00001400 in ASI 0x04. The normal read-only MFAR is at virtual address 0x00000400 in ASI 0x04.

**Note:**

SuperSPARC will never place instruction fault addresses in the MFAR. The information is not needed, since it is saved as the faulting PC/nPC when the trap occurs.

Only word-sized access to the MFAR is supported; other referenced sizes provoke a data\_access\_exception. The structure of the fault address register is as in Figure 9-13.

Figure 9–13. Fault Address Register

**Note:**

A rare scenario can occur when SuperSPARC is connected directly to the MBus with the store buffer off (not normal operating conditions). If a memory reference takes place that causes a copy-back, and the copy-back suffers a fault (memory fault), SuperSPARC responds by sending a `data_access_exception` to the pipeline. When this occurs, the MFAR holds the virtual address that caused the copy-out. Since this is not the address that caused the fault, SuperSPARC does not set MFSR.FAV.

If such an error occurs (MBus, `data_access_exception`, MFSR.FAV deasserted, store buffer disabled), system software can recover by forcing any modified data in the four possible cache lines (based on the virtual address) out to memory using transparent MMU references. Once this data has been flushed, the appropriate valid bits should be cleared, and the original operation may be retried. This situation is not expected to occur during normal operation.

**9.12.5 MMU Shadow FSR Register (MSFSR)**

This is identical to the MFSR but is used to record memory system errors while in emulation mode. This is done to avoid destroying the regular MFSR because of emulation instructions. This register is not used for normal operation. See Chapter 15 for more details.



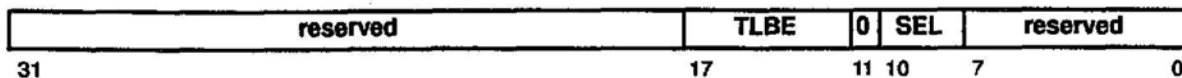
### 9.13 MMU Translation Look-Aside Buffer (TLB)

64 TLB entries, along with the root pointer and level-2 PTP, are accessible directly with the ASI value 0x06. This direct-access capability is provided for diagnostic purposes and also to lock TLB entries.

MMU entries are accessed as 32-bit words. Other data sizes provoke a `data_access_exception`.

The address format is shown in Figure 9-14.

Figure 9-14. MMU TLB Diagnostic Access Address



The various bit fields have the following meanings:

- reserved**      Reserved. All reserved bits are ignored (but should be zero).
- TLBE**          TLB Entry. Selects which of the 64 TLB entries is referenced.
- SEL**            Select. Determines what part of the referenced entry is accessed. The SEL field allows selection of either TLB fields or values cached in the root pointer and PTP2 caches. The entries in the sel field are described in Figure 9-15.

Figure 9-15. TLB Sel Field

Sel	Access
0	TLB Entry Virtual Page Number
1	TLB Entry Context Number
2	TLB Entry Physical Page Number
3	TLB Entry Lock Bit
4	Root Pointer
5	Level-2 PTP
6	Level-2 PTP Virtual Address
7	Reserved

For Sel values 4, 5, and 6, the TLBE field is not used. Data will be accepted or returned in the formats shown in Figure 9-16.

Figure 9-16. TLB Entry Virtual Page Number (SEL = 0)

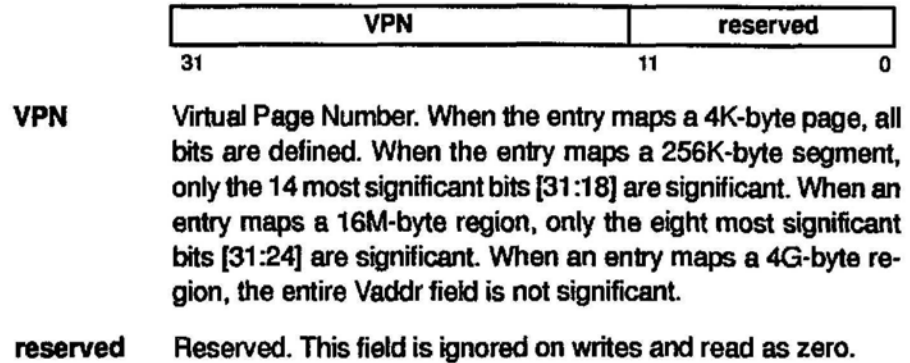


Figure 9-17. TLB Entry Context Number (SEL = 1)

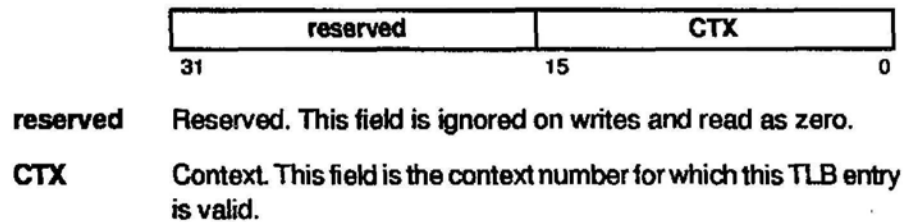
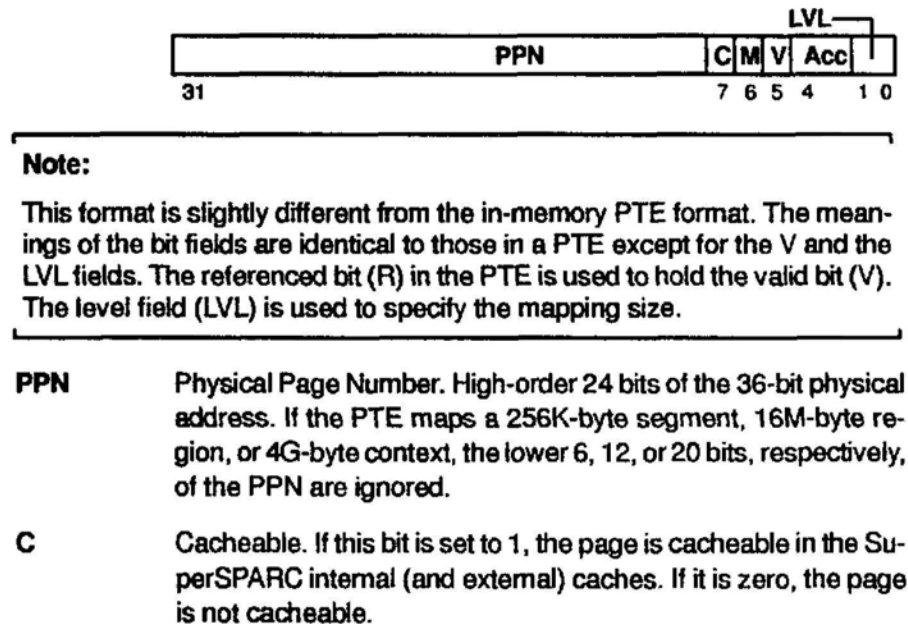


Figure 9-18. TLB Entry Physical Page Number (SEL = 2)



## MMU Translation Look-Aside Buffer (TLB)

<b>M</b>	Modified. When a page is accessed for writing and the modified bit is not set, the MMU sets the modified bit (M) in both the TLB and the main memory page table entry.
<b>V</b>	Valid. This field is set if the TLB entry is valid.
<b>ACC</b>	Access Permissions. This bit field encodes the access permissions associated with this TLB entry. The ACC field is encoded as shown in Table 9-1 or Table 9-16.
<b>LVL</b>	Level. This field is used to specify the mapping size as encoded in Table 9-18.

Table 9-18. Map Size Encoding

LVL	Mapping Size
0	4 Gigabytes
1	16 Megabytes
2	256 Kilobytes
3	4 Kilobytes

Figure 9-19. TLB Entry Lock Bit (SEL = 3)



<b>LOCK</b>	Lock Bit. If set to 1, the entry will not be displaced by the replacement algorithm. An entry can only be locked by storing a 1 in the lock bit; see Section 9.4. The Lock bit is set to 0 at power-on reset.
-------------	---

### Note:

If all TLB entries are locked, no replacement can take place, and a TLB miss will result in the processor entering an infinite table walk sequence. The processor will continue to read from the page tables forever and never return. No error will be reported, and the only way to exit is with reset.



**Note:**

It is possible to have multiple matches in the MMU if two or more entries mapping the same virtual space area are simultaneously present in the TLB. This cannot happen during the regular operating modes in which entries are loaded by the table-walking hardware. With the direct access capability, however, it is possible to erroneously load distinct entries mapping the same portion of a virtual space. When a virtual address from this area is translated by the MMU, the result is undefined. This condition is not reported to the software. Although operation is undefined, the hardware is internally protected and will not be damaged.

This condition may also occur under non-diagnostic situations if MMU demap transactions are not issued where required. As an example, if a normal level-3 PTE is present in the TLB, the page table is modified to include a level-2 or higher PTE mapping the same space, and a reference to a different location within the level-2 mapping. Under these conditions, the TLB will end up with two entries mapping the original page. A demap transaction is required after changing the page table mapping before any user instructions are generated. Demap operations are required by the reference MMU specification for these cases.

## **Caches/Store Buffer**

---

---

The SuperSPARC processor (SSP) has two large multi-way associative caches to provide high performance:

- ☐ 16K-byte data cache.
- ☐ 20K-byte instruction cache.

SuperSPARC also implements an eight-doubleword store buffer. The store buffer functions to hide most of the latency on writes to memory.

<b>Topic</b>	<b>Page</b>
10.1 Introduction .....	10-2
10.2 Instruction Cache .....	10-5
10.3 Instruction Cache Diagnostic and Control Interfaces .....	10-14
10.4 Data Cache .....	10-18
10.5 Data Cache Diagnostic and Control Interfaces .....	10-29
10.6 Store Buffer .....	10-34
10.7 Store Buffer Diagnostic and Control Interfaces .....	10-40

## 10.1 Introduction

The SSP contains a 20K-byte instruction cache memory and a 16K-byte data cache memory. Cache memories function by keeping copies of recently used data from main memory. Since the caches are small and close to the pipeline, they support high performance by supplying data or instructions much more quickly than is possible from main memory. They also have the secondary effect of reducing the number of transactions on the system bus, which is a benefit in multiprocessor systems.

The SSP's caches are organized with separate instruction and data caches, which is often called a Harvard architecture. The advantage of a Harvard architecture is that the two caches can be accessed at the same time, increasing the amount of work that the processor can accomplish in a cycle.

In configurations with the MultiCache Controller (MXCC), up to 2M-byte of external cache memory can be used. The organization, control, and programming of the external cache are described in Chapter 16.

The SSP also contains a store buffer that accepts stores, allowing the pipeline to continue processing instructions beyond the store. Stores in the buffer are passed to the bus when the bus is available and are removed from the buffer as they complete. The store buffer provides support for the total store ordering (TSO) and partial store ordering (PSO) memory models (see Chapter 8).

### **Cacheability**

Each load or store operation is either cacheable or non-cacheable, a difference that governs how the access is processed. Cacheable operations check the caches before accessing main memory or I/O devices, and may bring a copy of the data into the cache if it is not already there. Non-cacheable operations do not check the cache for the data before accessing main memory or I/O devices. Non-cacheable operations do not copy data into or out of the cache.

Generally, only cacheable accesses participate in cache coherence. In MBus systems, cacheable and non-cacheable accesses to the same address access the same memory location, while in XBus systems cacheable and non-cacheable address spaces generally do not overlap.

The cacheability of memory operations originating from an SSP is determined by the Memory Management Unit (MMU), according to control bits in the MCNTL register and the page table entry (PTE). There are three sources of memory references in the SSP: instruction fetches, load/store instructions, and the MMU. Table 10-1 summarizes the principal factors governing the cacheability of each.

Table 10–1. Cacheability Summary

Source	Cacheability Controls	Reference
Instruction Fetch	MCNTLEN MCNTLBT MCNTLIE MCNTLAC PTE.C	Table 10–2
Load/Store Instructions	MCNTLEN MCNTLDE MCNTLAC PTE.C	Table 10–5
MMU	MCNTLTC	

### Cache Consistency and Snooping

In multiprocessor systems, and even in uniprocessor systems with DMA I/O, it is important that accesses to a datum from different places in the system all get the same value for the datum. With caches, there may be several copies of a datum in different caches, in addition to the copy in main memory. Ensuring that all accesses see a consistent time series of values for a datum requires cache-consistency support in caches and on the system interconnect (bus).

In bus-based systems, all caches can easily observe all transactions on the shared system bus. In this environment, cache consistency protocols can be used that rely on each cache observing every bus transaction, even those for which it is neither the source nor the destination. This observing of the bus is called “bus snooping” and is critical to the cache-coherence protocols supported by the SuperSPARC processor.

The particular cache-coherence protocol used by the SuperSPARC chipset varies depending on the bus in use. For details related to the particular buses supported, see Chapter 17, Chapter 18, or Chapter 19.

The store buffer is a kind of cache and must snoop bus transactions. Data in the store buffer has been modified but not yet communicated to the rest of the system. If another processor’s cache attempts to access a datum that is contained in the store buffer, the buffer must act as the owner of the data and supply it.



### ***Consistency and Snooping Between Instruction Cache, Data Cache, and Store Buffer***

The two internal caches and the store buffer all snoop internal as well as external transactions. Usually this snooping occurs on the bus and is visible on the bus pins of the SSP. For example, in the direct MBus configuration, if the instruction cache attempts to read a sub-block that is dirty in the data cache, the read will appear on MBus, and the data cache will assert MIF on MBus to indicate that it will supply the data.

The store buffer snoops data cache transactions and will hold the data cache if it attempts access to data still in the store buffer until the store has been acknowledged from main memory.

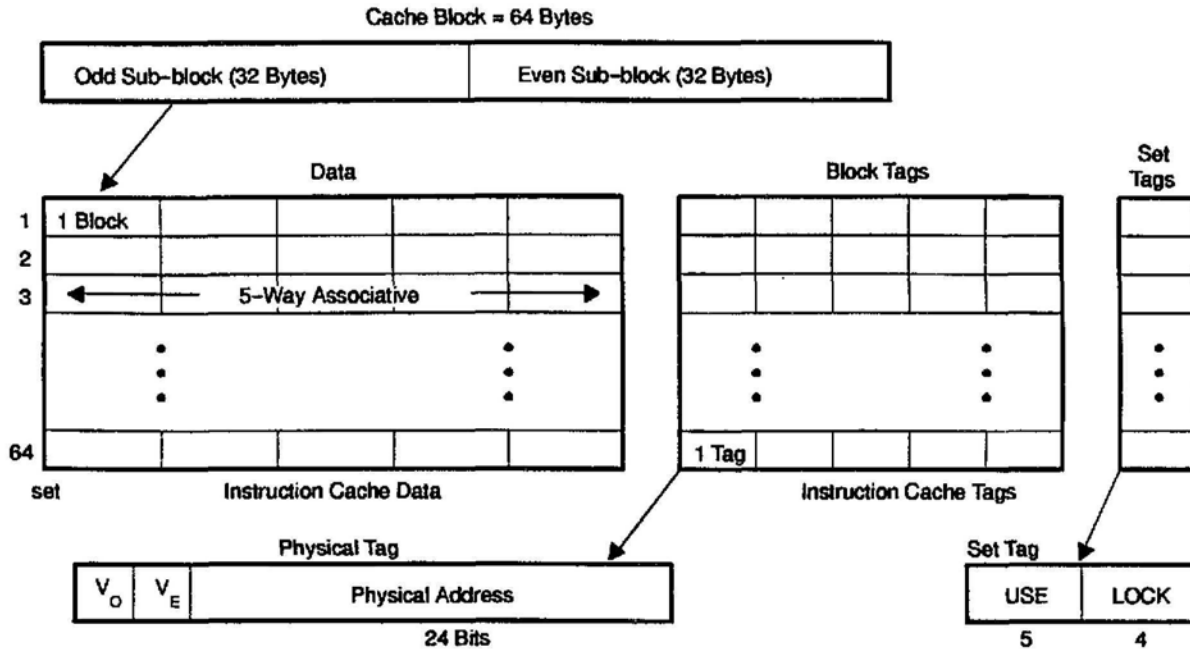
## **10.2 Instruction Cache**

The SuperSPARC processor has an internal instruction cache to support high-performance execution. It features:

- ☐ 20K-byte total capacity.
- ☐ Five-way set-associative organization with 64 sets.
- ☐ Tags containing physical addresses.
- ☐ 64-byte block size.
- ☐ Blocks divided into 32-byte sub-blocks.
- ☐ 128-bit (four-instruction) access for instruction fetch.
- ☐ Prefetching.

The instruction cache is organized as shown in Figure 10-1. The cache is five-way set associative with 64 sets. Physical address bits PA[11:6] select one of the 64 sets. Each set stores five blocks. Each set has a set tag (STag) that contains the usage and lock information for the five blocks in the set (see Figure 10-6).

Figure 10–1. Instruction Cache Organization



Each of the five blocks in a set has a physical tag (PTag) and two sub-blocks that hold the cached instructions. The PTag contains the upper 24 bits of the physical address of the data currently contained in the block, if any, and a valid bit for each of the sub-blocks. The even sub-block stores data with PA[5]=0, and the odd sub-block stores data with PA[5]=1. See Figure 10–5 for the PTag format.

On an instruction fetch, if a valid block is found in the instruction cache for the fetch address, the cache will send up to 128 bits (four sequential instructions) to the instruction queue (see Table 10–2). The number of instructions sent will be limited by the block boundary or the sub-block boundary if the odd sub-block is invalid.

The cache is enabled by the MCNTLIE bit of the MMU control register (see Subsection 9.12.1). The cache is disabled at power-on (see Section 13.2, Hardware Reset) and remains disabled until the MCNTLIE bit is set.

At power-on the contents of the instruction cache are undefined. It is the responsibility of the software to initialize the instruction cache by resetting the valid bits (see Subsection 10.3.1).

### 10.2.1 Instruction Cache Access

These are the steps that compose an instruction cache access:

- 1) The virtual address is translated by the MMU to produce a physical address, PA[35:0]. An instruction\_access\_exception may be raised by the MMU.
- 2) PA[11:6] selects one of the 64 sets.
- 3) The address in the PTags of the five blocks in the set are compared to PA[35:12], ignoring any invalid sub-blocks.
- 4) If there is a matching block address and the sub-block selected by PA[5] is valid in the PTag, the block is selected. Otherwise, the access is a cache miss, and the sub-block is read from memory.
- 5) PA[5:2] selects the first four-byte instruction from the matching sub-block. It is aligned along with other sequentially successive instructions within the valid sub-blocks of the selected cache block. The resulting aligned instructions, up to four instructions (128 bits), are sent to the instruction queue.

### 10.2.2 Cacheability

Whether the instruction fetches are cached depends on the operating mode of the MMU, as set in the MMU control register (MCNTL), and cacheable bits within each page table entry (PTE). See Table 10-2.

Table 10–2. Instruction Cacheability

Translation Mode	MCNTL.IE=0 MCNTL.AC=X	MCNTL.IE=1 MCNTL.AC=0	MCNTL.IE=1 MCNTL.AC=1
Boot Mode MCNTL.BT=1 MCNTL.EN=X	Not Cached	Not Cached	Not Cached
MMU Disabled MCNTL.BT=0 MCNTL.EN=0	Not Cached	Not Cached	Cached
MMU Enabled MCNTL.BT = 0 MCNTL.EN = 1	Not Cached	C=1: Cached	C=1: Cached
		C=0: Not Cached	C=0: Not Cached

BT = Boot Mode bit of the MMU control register.  
 EN = MMU Enable bit of the MMU control register.  
 IE = Instruction Cache enable bit of the MMU control register.  
 AC = Alternate Cacheable bit of the MMU control register.  
 C = Cacheable bit kept in each Page Table Entry.  
 X = "don't care"

The entries in the table labeled "Not Cached" mark cases where the instruction cache is not accessed and instructions read from memory are not entered into the instruction cache. Entries marked "Cached" are for cases where instructions read from memory are entered into the instruction cache, and the cache supplies the instructions, if they are present.

### 10.2.3 Instruction Cache Miss Processing

Miss processing is caused when a demand fetch fails to match a valid sub-block tag in the cache. Prefetches (see Subsection 10.2.4) never provoke miss processing.

In miss processing, the required sub-block is read in from the next level of the memory hierarchy and forwarded to the instruction queue. Simultaneously, the new sub-block is placed in the cache. This may require that a valid block be displaced from the cache. Since the instruction cache is five-way set-associative, there are five candidates for replacement. The choice among the candidates is controlled by the replacement policy.

### ***Instruction Cache Replacement Policy***

A limited-history algorithm determines which blocks in the cache will be replaced. The STag contains a usage bit for each block of the set and a lock bit for all but block 0. Whenever an instruction fetch hits in the cache, the usage bit is set. If all the other usage bits in the STag are already set, they are all cleared. All five bits are never set at the same time, and the usage bit of the most recently used block will always be set. When the usage bits are reset, the history begins accumulating again. In this way, a limited record of the most recently used members of a set can be kept.

The maintenance of the usage bits is modified by the lock bits. For the purpose of usage bit updates, the usage bit for a locked block is treated as if it were always set.

A block that is locked into the cache is never selected for replacement. Block 0 cannot be locked; therefore, if all other entries are locked, it will always be selected for replacement, regardless of the state of the usage bits.

If there is more than one unlocked block, the usage bits determine which block should be replaced. If there is an unlocked block with a cleared usage bit, it is a candidate for replacement. A candidate is selected for replacement according to a fixed priority order. Block 4 has the highest priority order, and block 0 the lowest. The instruction cache therefore fills starting at block 4, then progresses to 3, 2, 1, and finally 0.

Case One and Case Two of Example 10-1 illustrate two replacement schemes.

**Example 10–1. Instruction Cache Replacement**

**Case One**

Block	4	3	2	1	0
USE	1	1	0	0	1
LCK	0	0	0	1	
Replacement Candidate	No	No	Yes	No	No

**Case Two**

Block	4	3	2	1	0
USE	1	0	0	1	1
LCK	0	0	0	0	
Replacement Candidate	No	Yes	Yes	No	No

USE = Usage - STag bits providing a limited history used bits.

LCK = Lock - STag bits providing information on locked blocks.

In Example 10–1, Case One, based on the replacement candidate line, block 2 is chosen for replacement. In Case Two, block 3 is chosen, since it is the left-most available block. Note again that block 0 can never be locked. This is to ensure that cacheable data may always be stored in the cache (when the cache is enabled).

**Snoop Hits and Lock Bits**

Cache-consistency transactions use physical addresses, so the PTags are consulted for address comparison. A snoop hit occurs when a cacheable bus transaction matches the PTag for a block in the cache. A block will be invalidated (valid bits cleared) when there is a snoop hit on VBus or an invalidation transaction on MBus. Invalidation transactions include coherent read invalidate (CRI) (CRI), coherent invalidate (CI), and coherent write invalidate (CWI).

If a block is invalidated and the lock bit is set, the lock bit will not be cleared. This prevents the address cache block from being used, thus reducing the number of active cache blocks by 1.

**Note:**

If a block is to be locked, ensure that it will not be invalidated.

See Subsection 10.2.5 for further information on cache consistency operations in the instruction cache.

### ***Instruction Cache Miss Penalties and Timing***

When an instruction cache miss occurs, a number of events occur that can affect instruction processing timing. First, since the instruction cache does not contain the requested instruction, there is no instruction to send to the instruction queue. If the request was a demand fetch (not a prefetch), the instruction queue will become empty, and bubbles (empty instruction groups) will be issued into the pipeline until the instruction is available for decoding. Even if the fetch request was for a prefetch, it is likely that the instruction queue will become empty before the miss is serviced from memory.

Once a miss is detected, the instruction cache arbitrates for the bus interface (other requesters might be snoop requests, data cache bus accesses, the store buffer, and MMU bus accesses). Once the bus interface is available, arbitration is required again, this time to acquire access to the bus. The method of arbitration depends on the bus in use, either MBus (see Chapter 17) or VBus (see Chapter 18). After access to the bus has been obtained, the requested sub-block is read. Once the data returns, it is sent to the instruction queue and the cache simultaneously.

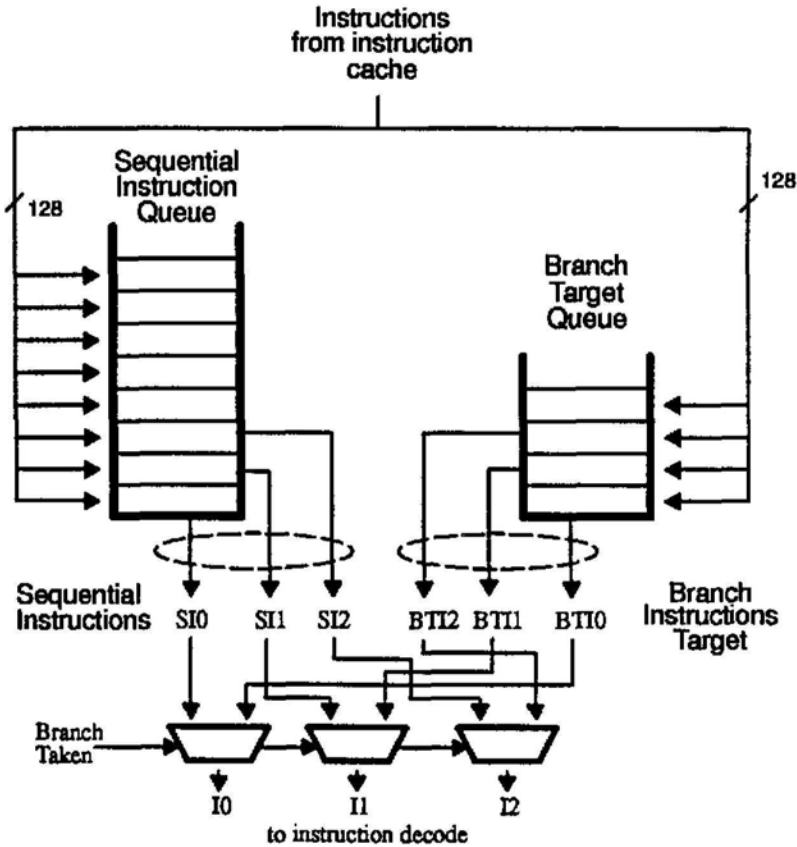
When the data for an instruction cache miss is found in the external cache (VBus configuration), it takes a minimum of five cycles more than an instruction cache hit.

#### **10.2.4 Instruction Prefetching and the Instruction Queue**

The instruction prefetcher supplies instructions directly to the Instruction Queue (IQ) without buffering. The IQ comprises an eight-word first-in, first-out (FIFO) sequential instruction queue and a four-word branch target queue (see Figure 10-2). The IQ provides instructions for the pipeline to execute. The pipeline can extract up to three instructions per cycle from the IQ. The prefetcher continuously tries to fill the IQ with the instructions, either from the instruction cache (on a cache hit) or from memory (on a cache miss), according to the next sequential address location. When a control transfer instruction (CTI) is encountered, the prefetcher immediately retrieves the target instructions and places them in the branch target queue. If the CTI is taken, the pipeline will extract instructions from the branch target queue; if the CTI is not taken, the branch target queue will be ignored, and fetching will continue from the sequential instruction queue.



Figure 10–2. Instruction Queue



A demand fetch is a memory access that occurs when the processor needs instructions that are not currently available in the IQ. This can occur due to changes in the program counter during exceptions, traps, or CTIs. It also occurs when an instruction is required before it has been delivered to the IQ.

Only a demand fetch—not a prefetch—can initiate an MMU table walk or cause an exception. If a required translation is not present in the translation lookaside buffer (TLB), the prefetch logic will abandon the prefetch attempt. A demand fetch may be required later for the same address translation, and the MMU will perform the table walk at this time.

Instruction prefetching is always enabled when the instruction cache is on. There is no explicit control bit. Instruction prefetching for SuperSPARC works identically on VBus and MBus. Errors may occur during prefetching; see Section 8.6 for details on prefetch exception handling.

### 10.2.5 Instruction Cache Consistency

The instruction cache snoops transactions on MBus or VBus and is fully consistent with the internal data cache and all external caches obeying the cache-consistency protocol for that bus. Instruction cache consistency is maintained in hardware.

Cache-consistency transactions use physical addresses, so the PTags are consulted for address comparison. A snoop hit occurs when a cacheable bus transaction matches the PTag for a block in the cache. A block will be invalidated (valid bits cleared) when there is a snoop hit on VBus or an invalidation transaction (i.e., CRI, CI, CWI) on MBus.

The instruction cache does not allow writes, so it never becomes an owner of data, and it never needs to transmit its contents back to the system bus. All instruction cache snoop hits are handled by invalidating the appropriate cache entries. This includes snoop hits by transactions generated by load and store operations on the same processor. The invalidations are executed as long as the MCNTL.SE (snoop enable) bit is set.

#### *Self-Modifying Code*

The instruction cache cannot be written directly. Instead, the normal cache-consistency mechanism is used between the instruction cache and the data cache. When a store instruction is attempted to an address that is present in the instruction cache, the data cache must obtain ownership of the block before the store can proceed. In order to acquire ownership, any copies of the block that exist in any of the other caches in the system must be invalidated. The instruction cache will see this invalidation due to its snooping of all bus activity, and it will invalidate its copy of the block. Should the instruction cache attempt to fetch from the block after the invalidation, the cache-consistency protocol for the bus will ensure that the instruction cache receives the new data from the data cache.

A FLUSH instruction must be performed to ensure instruction cache consistency when you execute self-modifying code. (See Section 7.4 for more information on FLUSH instructions.) The FLUSH instruction forces the execution of all the pending writes and will also flush the instruction prefetch buffer and pipeline. The FLUSH instruction executes synchronously, implying that the SuperSPARC processor is stalled until all previous memory operations are completed. The instruction that was modified prior to the FLUSH instruction is guaranteed to execute properly after the FLUSH has been completed.

### 10.3 Instruction Cache Diagnostic and Control Interfaces

This section describes the low-level diagnostic and control interfaces to the instruction cache. These interfaces are accessed via LDA, LDDA, STA, and STDA instructions to an ASI assigned for this purpose. Table 10-3 shows the ASIs used for diagnostic and control access to the instruction cache.

Table 10-3. Instruction Cache ASIs

Function	ASI
Instruction Cache Flash Clear	0x36
Instruction Cache Tags	0x0c
Instruction Cache Data	0x0d

#### 10.3.1 Instruction Cache Flash Clear (ASI=0x36)

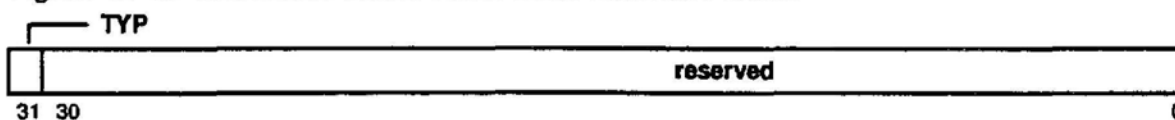
You can invalidate the entire instruction cache or clear all the lock bits by issuing a store alternate with the ASI value 0x36. The data size of the store operation must be a 32-bit word. All other data sizes cause a `data_access_exception`. The data issued by the store operation is ignored. The most significant bit of the address determines the type of operation.

**Note:**

Flash clear operations should always be executed before enabling the instruction cache.

The address format is shown in Figure 10-3.

Figure 10-3. Instruction Cache Flash Clear Address Format



Bit field explanations:

TYP	Type of operation.
0:	Invalidate. Both valid bits in the PTags of each block in the instruction cache are cleared; all USE bits are cleared in the STags of each set in the instruction cache. The Lock bits of the STags are not affected.
1:	Unlock. All Lock bits in the STags of each set in the instruction cache are cleared.
reserved	Reserved. These bits are ignored.

### 10.3.2 Instruction Cache Tags (ASI=0x0c)

Instruction cache tags are readable and writable with ASI value 0x0c. This direct-access capability is provided for diagnostic purposes.

A PTag) is associated with each cache block, and the STag is associated with each set. The PTags and the STags must be accessed as 64-bit doublewords. All other data sizes cause a `data_access_exception`. The contents of the cache tags are not affected by watchdog reset or hardware reset.

The tags are addressed as pairs because each instruction cache block has both a PTag and an STag. The address format is as follows:

Figure 10–4. Instruction Cache Tag Address Format

TYP	r	BLK	res								SET		res		zero		
31	30	29	28	26	25					12	11		6	5	3	2	0

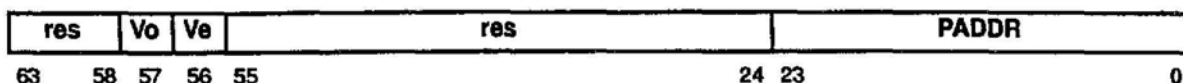
The fields of data cache tag address are:

<b>TYP</b>	Type of operation.
0:	reserved. Generates a <code>data_access_exception</code> .
1:	STag. Access the set tag. BLK is ignored. SET selects the set to access.
2:	PTag. Access the physical tag. SET and BLK are used to access a particular block's PTag.
3:	reserved. Generates a <code>data_access_exception</code> .
<b>r, res</b>	Reserved. These bits are ignored.
<b>BLK</b>	Block. Selects one of the five blocks in a set (0–4). Blocks 5 – 7 do not exist. Attempting to access one of blocks 5–7 generates a <code>data_access_exception</code> .
<b>SET</b>	Set selects one of the instruction cache's 64 sets to access.
<b>zero</b>	Zero field. This field should always be zero.

#### PTag Diagnostic Access

Figure 10–5 shows the instruction cache PTag diagnostic access format.

Figure 10–5. Instruction Cache PTag Diagnostic Access Format



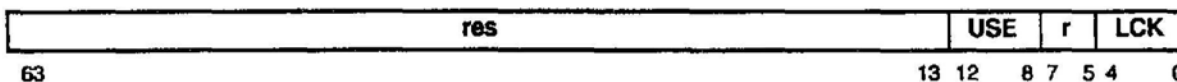
Bit field explanations:

- res** Reserved. Read as 0 and ignored on a write.
- Vo** Valid bit for the odd sub-block. When this bit is cleared, the odd sub-block is invalid. The odd sub-block stores data at addresses with PA[5]=1. At power-on (see Section 13.2), this bit is undefined.
- Ve** Valid bit for the even sub-block. The even sub-block stores data at addresses with PA[5]=0. At power-on (see Section 13.2), this bit is undefined.
- PADDR** Physical Address Tag. Bits [35:12] of the physical address of the data contained in this cache block. Note that if neither Vo or Ve is set, the PTag has no meaning. If either or both of Vo and Ve is set, the PADDR field is valid.

### STag Diagnostic Access

Figure 10–6 shows the instruction cache STag diagnostic access format.

Figure 10–6. Instruction Cache STag Diagnostic Access Format



Bit field explanations:

- res, r** Reserved. Read as 0 and ignored on a write.

**USE** Usage bits. This bit field indicates blocks of the set that have been accessed recently. The position of the bit in the field determines which block it is associated with. Bits 8–12 correspond to blocks 0–4, respectively. This bit field is updated by the hardware replacement algorithm and is used to select a block for replacement.

USE4	USE3	USE2	USE1	USE0
12	11	10	9	8

**LCK** Lock bits. Each lock bit can lock a block inside the cache. The position of the bit determines which block it locks. Bit 0 is fixed to 0 and ignores write. Bits 1–4 lock blocks 1–4, respectively. When a lock bit is set to 1, the corresponding block will not be displaced by the replacement algorithm.

LCK4	LCK3	LCK2	LCK1	0
4	3	2	1	0

### 10.3.3 Instruction Cache Data (ASI=0x0d)

Instruction cache data is readable and writable with ASI value 0x0d. This direct-access capability is provided for diagnostic purposes. The blocks must be accessed as 64-bit doublewords. Other data sizes cause a `data_access_exception`. Instruction cache data is not affected by watchdog reset, hardware reset, or flash clear. The address format is shown in Figure 10–7.

Figure 10–7. Instruction Cache Data Address Format

res	BLK	res	SET	DWORD	zero
31 29 28 26 25		12 11	6 5	3 2	0

The fields of an instruction cache data address are:

**res** Reserved. These bits are ignored.

**BLK** Block. Designates which of the instruction cache's five blocks (0–4) are referenced. Blocks 5–7 do not exist. Attempting to access one of blocks 5–7 generates a `data_access_exception`.

**SET** Select. Selects one of 64 sets of the cache.

**DWORD** Doubleword. Selects the doubleword within the cache block.

**zero** Zero field. This field should always be zero.

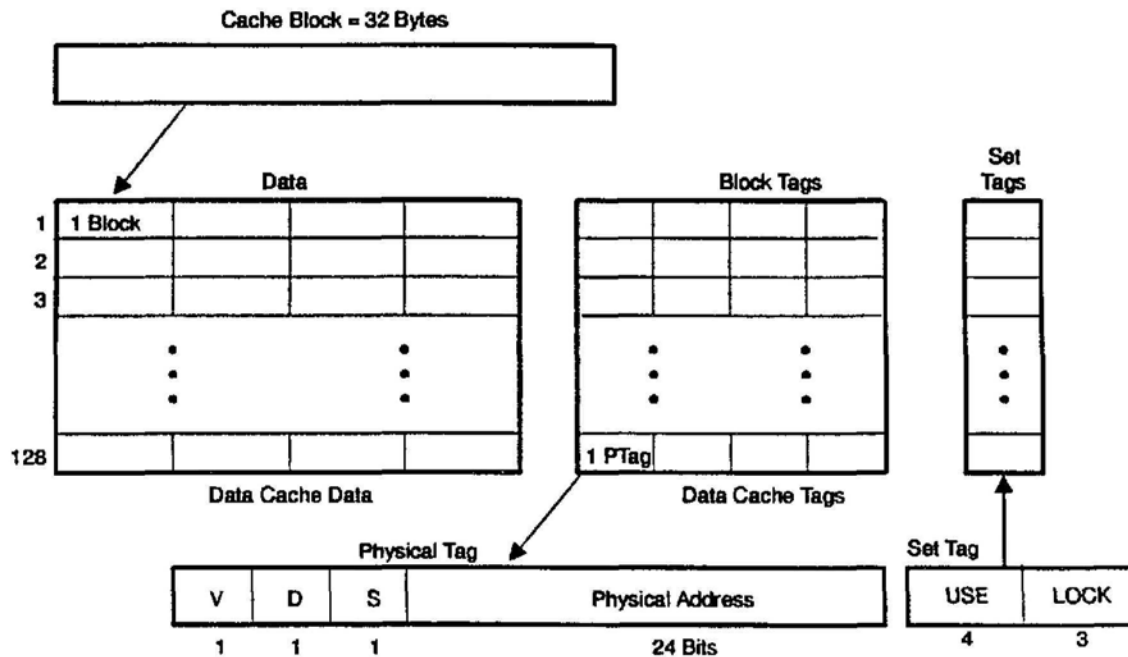
## 10.4 Data Cache

The SuperSPARC processor has an internal data cache that features:

- ☐ 16K-byte total capacity.
- ☐ Four-way set-associative organization with 128 sets.
- ☐ Tags containing physical addresses.
- ☐ 32-byte block size.
- ☐ No sub-blocks.
- ☐ Up to 64-bit data access.

The data cache organization is shown in Figure 10-8. The cache is four-way set associative with 128 sets. Physical address bits PA[11:5] select one of the 128 sets. Each set stores four blocks. Each set has an STag that contains the usage and lock information for the four blocks in the set (see Figure 10-8).

Figure 10-8. Data Cache Organization



Each of the four blocks in a set has a PTag and a block of 32 bytes that holds cached data. The PTag contains the upper 24 bits of the physical address of the data currently contained in the block, if any, and a valid bit that indicates whether the block contains any valid data. The PTag also has shared (S) and dirty (D) bits that are used by the cache-consistency protocols.

The cache is enabled by the MCNTLDE bit (see Subsection 9.12.1). The cache is disabled at power-on reset and remains disabled until the MCNTLDE bit is set.

At power-on, the contents of the data cache are undefined. It is the responsibility of the software to initialize the data cache by using data cache flash clear to reset the valid bits. After a watchdog reset, the contents of the data cache are unmodified.

#### **10.4.1 Data Cache Access**

The steps in a data cache access are:

- 1) The virtual address is translated by the MMU to produce a physical address, PA[35:0]. A data\_access\_exception may be raised by the MMU.
- 2) PA[11:5] selects one of the 128 sets.
- 3) The four block tags (PTags) in the set are compared to PA[35:12], ignoring any invalid blocks.
- 4) If there is a matching block, it is selected. Otherwise, the access is a cache miss, and the block is read from memory.
- 5) PA[4:0] and the size of the transaction determine which bytes are returned, as shown in Table 10-4.



Table 10—4. Data Bytes of the Data Cache Block Returned from Access

PA[4:0]	Access Size			
	Byte	16-bit Halfword	32-bit Word	64-bit Doubleword
0	0	0:1	0:3	0:7
1	1	✱	✱	✱
2	2	2:3	✱	✱
3	3	✱	✱	✱
4	4	4:5	4:7	✱
5	5	✱	✱	✱
6	6	6:7	✱	✱
7	7	✱	✱	✱
8	8	8:9	8:11	8:15
9	9	✱	✱	✱
10	10	10:11	✱	✱
11	11	✱	✱	✱
12	12	12:13	12:15	✱
13	13	✱	✱	✱
14	14	14:15	✱	✱
15	15	✱	✱	✱
16	16	16:17	16:19	16:23
17	17	✱	✱	✱
18	18	18:19	✱	✱
19	19	✱	✱	✱
20	20	20:21	20:23	✱
21	21	✱	✱	✱
22	22	22:23	✱	✱
23	23	✱	✱	✱
24	24	24:25	24:27	24:31
25	25	✱	✱	✱
26	26	26:27	✱	✱
27	27	✱	✱	✱
28	28	28:29	28:31	✱
29	29	✱	✱	✱
30	30	30:31	✱	✱
31	31	✱	✱	✱

✱ marks illegal address alignment. These combinations of address and access size provoke a memory\_address\_not\_aligned exception.

n numbers indicate the byte within the cache block.

n:n number ranges indicate the range of bytes within the cache block.

### 10.4.2 Write-Through or Copy-Back Cache

The MCNTL.MB bit is a read-only indicator of whether the SuperSPARC processor is operating directly on the MBus or whether it is connected to the MXCC over VBus. This bit reflects the state of the CCMODE pin at reset. When SuperSPARC is used with the MXCC, the data cache operates as a write-through cache. When CCMODE is low at reset, the processor uses VBus, and the data cache operates as a write-through cache. When the processor is operating directly on the MBus with no MXCC, the data cache operates as a copy-back cache with write allocation.

When the processor is used on the VBus, the data cache is operated as a write-through cache, and all stores are immediately written to the store buffer. The store buffer will send them out to the external cache and main memory as soon as resources are available. Writes do not allocate a block in the data cache if it is not already present.

When the processor operates directly on the MBus, cache blocks that have been modified by the processor are written back to memory only when necessary because of replacement or snoop reads. The data cache operates with write-allocation: when a store miss occurs, the data cache will allocate a block, bring in the missing data from memory, and write that data into the cache block.

### 10.4.3 Cacheability

Data is cached depending on the mode set within the MMU control register (MCNTL) and cacheable bits within each PTE, as shown in Table 10-5.

Table 10-5. Data Cacheability from Loads and Stores

Translation Mode	MCNTL.DE=0 MCNTL.AC=X	MCNTL.DE=1 MCNTL.AC=0	MCNTL.DE=1 MCNTL.AC=1
MMU Transparent	Not Cached	Not Cached	Cached
MMU Disabled MCNTL.BT=X MCNTL.EN=0	Not Cached	Not Cached	Cached
MMU Enabled	Not Cached	C=1: Cached	C=1: Cached
MCNTL.BT = X MCNTLEN = 1		C=0: Not Cached	C=0: Not Cached

BT = Boot Mode bit of the MMU control register.  
 EN = MMU Enable bit of the MMU control register.  
 DE = is the Data Cache enable bit of the MMU control register.  
 AC = Alternate Cacheable bit of the MMU control register.  
 C = Cacheable bit kept in each Page Table Entry.  
 X = "don't care".

The entries in the table labeled "Not Cached" mark cases where the data cache is not accessed and data read from memory is not entered into the cache. Entries marked "Cached" are for cases where the data cache supplies the data if it is present in the cache and, if not present, data read from memory is entered into the cache.

Generally, references are non-cacheable when the data cache is disabled. When the cache is enabled, the C bit from the PTE controls cacheability. For special accesses (such as MMU pass-through/bypass transactions) without a corresponding C bit, the AC (alternate cacheable) bit is used to determine cacheability in the MCNTL register.

The cacheability of data in the internal data cache also controls cacheability in the external cache. For MMU table walk references, the MCNTL.TC (table walk cacheable) bit indicates external cacheability, even though table walk data is never cached internally.

### 10.4.4 Data Cache Miss Processing

A data cache miss occurs when PA[35:12] does not match the PADDR field of the PTag for any of the four blocks in the set selected by PA[11:5]; the matching PTag must also be valid (have V set). Miss processing is the series of actions triggered by a cache miss.

To process a data cache miss, the required block is read in from the next level of the memory hierarchy and placed in the cache. This may require that a valid block be displaced from the cache. Since the data cache is four-way set-associative, there are four candidates for replacement. The choice among the candidates is controlled by the replacement policy.

#### *Data Cache Replacement Policy*

A limited-history algorithm determines which blocks in the cache will be replaced. The STag contains a usage bit for each block of the set and a lock bit for all but block 0. Whenever a data reference hits in the cache, the usage bit is set. If all the other usage bits in the STag are already set, they are all cleared. All four bits are never set at the same time, and the usage bit of the most recently used block will always be set. When the usage bits are reset, the history begins accumulating again. In this way, a limited record of the most recently used members of a set can be kept.

The maintenance of the usage bits is modified by the lock bits. For the purpose of usage bit updates, the usage bit for a locked block is treated as if it were always set.

A block that is locked into the cache is never selected for replacement. Block 0 cannot be locked; therefore, if all other entries are locked, it will always be selected for replacement, regardless of the state of the usage bits.

If there is more than one unlocked block, the usage bits determine which block should be replaced. If there is an unlocked block with a cleared usage bit, it is a candidate for replacement. A candidate is selected for replacement according to a fixed priority order. Block 3 has the highest priority order, and block 0 the lowest. The data cache therefore starts filling at block 3, then progresses to 2, 1, and finally 0.

Cases One and Two of Example 10-2 illustrate two replacement schemes.

### Example 10-2. Data Cache Replacement

#### Case One

Block	3	2	1	0
USE	1	0	0	1
LCK	0	0	1	
Replacement Candidate	No	Yes	No	No

#### Case Two

Block	3	2	1	0
USE	0	0	1	1
LCK	0	0	0	
Replacement Candidate	Yes	Yes	No	No

USE = Usage - STags providing a limited history used bits.

LCK = Lock - STags providing locked blocks information.

In Example 10-2, Case One, based on the replacement candidate line, block 2 is chosen for replacement. In Case Two, block 3 is chosen, since it is the left-most available block. Note again that block 0 can never be locked. This is to ensure that cacheable data may always be stored in the cache (when the cache is enabled).

## Snoop Hits and Lock Bits

Cache-consistency transactions use physical addresses, so the PTags are consulted for address comparison. A snoop hit occurs when a cacheable bus transaction matches the PTag for a valid block in the cache. A block will be invalidated (valid bits cleared) when there is a snoop hit on VBus or an invalidation transaction (i.e., CRI, CI, CWI) on MBus.

If a block is invalidated and the lock bit is set, the lock bit will not be cleared. This prevents the cache block from being used, thus reducing the number of active cache blocks by 1.

---

**Note:**

If a block is to be locked, ensure that it will not be invalidated.

---

See Subsection 10.4.6 for further information on cache-consistency operations in the data cache.

## 10.4.5 Data Cache Miss Timing

The timing for a data cache miss depends on the bus in use and whether a block in the data cache needs to be replaced. The following summarizes several of the common cases. None of the timings given accounts for any delays in obtaining access to the SSP's bus interface or to the bus itself, so all should be considered minimum delays.

### *VBus with External Cache*

Following is the sequence for a data cache miss in the VBus with external cache configuration. No block replacement is needed because internal data cache blocks are never dirty.

- 1) LD instruction accesses TLB and data cache. The cache miss is detected. This cycle is where the data would be accessed for a cache hit.
- 2) LD instruction frozen in E0. This is the first extra clock cycle.
- 3) VBus command word cycle. The SSP issues a 32-byte read on VBus. The external cache SRAMs are pipelined. Like MXCC, they latch the address on this cycle.
- 4) The external-cache SRAMs are pipelined. SRAMs work internally during this cycle, using the address latched from the previous cycle. MXCC also determines external cache hit or miss during this cycle.

- 5) The external cache SRAMs return the first doubleword in this cycle.
- 6) In the following cycle, the data cache writes the first doubleword of data into the cache and also sends the data to the LD instruction, which completes with its WB stage.

So, when a load misses in the internal cache, there is a five-cycle stall for data from the second-level cache. Stores do not write-allocate when external cache is used and proceed immediately to the store buffer without any stalls.

The next three doublewords of data are returned after the LD instruction has been satisfied and while the pipeline proceeds with other instructions. The additional doublewords of the cache block keep the VBus busy for three additional cycles.

See Chapter 18 for more information on bus cycles in this configuration.

### ***Direct MBus When No Block Replacement Needed***

For a data cache miss in the direct MBus configuration when no block replacement is needed (the block is not dirty), the sequence is:

- 1) LD instruction accesses TLB and data cache. The cache miss is detected. This cycle is where the data would be accessed for a cache hit.
- 2) LD instruction frozen in E0. This is the first extra clock cycle.
- 3) MBus command word cycle. The SSP issues a 32-byte coherent read on MBus.
- 4) The memory returns the first doubleword. The number of cycles after the command word cycle depends on the system design and memory speed.
- 5) In the following cycle, the data cache writes the first doubleword of data into the cache and also sends the data to the LD instruction, which completes with its WB stage.

So the miss penalty for MBus when no replacement is needed is three cycles plus the memory system's command word to first data delay.

The next three doublewords of data are returned after the LD instruction has been satisfied and while the pipeline proceeds with other instructions. The additional doublewords of the cache block keep the MBus busy for additional cycles.

### ***MBus With Block Replacement***

For a data cache miss in the direct MBus configuration when block replacement is needed (the block is marked dirty), the sequence is:

- 1) LD instruction accesses TLB and data cache. The cache miss is detected. This cycle is where the data would be accessed for a cache hit.
- 2) LD instruction frozen in E0. This is the first extra clock cycle. In this cycle, the SSP also issues a command word cycle on MBus to start a 32-byte write.
- 3) In the next four cycles, the SSP sends the four doublewords of the replaced block to memory via MBus.
- 4) The SSP may have to wait for all doublewords to be acknowledged on MBus.
- 5) MBus is idle for one cycle after the last doubleword of the 32-byte write is acknowledged.
- 6) The SSP issues a command word cycle on MBus for a 32-byte coherent read.
- 7) The memory returns the first doubleword. The number of cycles after the command word cycle is dependent on the system design and memory speed.
- 8) In the following cycle, the data cache writes the first doubleword of data into the cache and also sends the data to the LD instruction, which completes with its WB stage.

The miss penalty for MBus when replacement is needed is eight cycles plus the memory system's command word to first data delay and any delays that the memory controller inserts into the burst write.

The next three doublewords of data are returned after the LD instruction has been satisfied and while the pipeline proceeds with other instructions. The additional doublewords of the cache block keep the MBus busy for additional cycles.

### **10.4.6 Data Cache Consistency**

In a multiple-processor system, a mechanism must exist to keep local caches consistent with each other and with main memory. The SuperSPARC processor uses a protocol for this purpose that is implemented in hardware. Part of this protocol involves snooping bus transactions. The purpose of snooping is to ensure that the contents of the data cache are consistent with external caches and main memory. Both the data cache and the store buffer snoop for incoming transactions that request data invalidation.

All addresses that are snooped are compared with the cache tags in the instruction cache and the data cache. Cache-consistency transactions use physical addresses, so the PTags are consulted for address comparison. A snoop hit occurs if the address presented on the bus matches a valid PTag in the cache set to which the address belongs. Cache snooping is controlled by MCNTL.SE—snoop-enable.

The action taken on a snoop hit depends on whether the SSP is being used with the MXCC on VBus or is connected directly on MBus. When SuperSPARC is used with the MXCC, preserving this consistency is fairly simple; an entry in the cache is invalidated if some other processor writes to this location. Chapter 18 explains the VBus protocol.

For SuperSPARC on MBus, the actions taken on a snoop hit are more complex and may include responding to the bus transaction with the current cache contents. The data cache responses on external transactions are listed in Table 10-6. Chapter 17 has a more complete description of the MBus cache-consistency protocol.

*Table 10-6. Data Cache Snoop Mechanism (MBus)*

External Transaction Being Snooped	Response in Cache	Resulting Action
R, W	Don't Care	No action
CR, CRI, CI, CWI	Miss	No action
CR	Hit, not owned, not shared	Set shared bit
CR	Hit, not owned, shared	No action
CR	Hit, owned, not shared	Copy out data, set shared bit
CR	Hit, owned, shared	Copy out data
CRI, CI, CWI	Hit, not owned, not shared	Invalidate entry
CRI, CI, CWI	Hit, not owned, shared	Invalidate entry
CRI	Hit, owned, not shared	Copy out data, Invalidate entry
CWI, CI	Hit, owned, not shared	Invalidate entry
CRI	Hit, owned, shared	Copy out data, Invalidate entry
CI, CWI	Hit, owned, shared	Invalidate entry

R= Read(non coherent)

W=Write (non-coherent)

CR=Coherent Read

CI=Coherent Invalidate

CRI=Coherent Read and Invalidate

CWI=Coherent Write and Invalidate

(Refer to the *SPARC MBus Specification* for definitions of these terms).



Table 10-6 shows the responses of the data cache to various MBus transactions, depending on the state of the block in the cache that is accessed. This status is held in the cache tag in the shared and dirty bits.

---

**Note:**

The dirty bit being set indicates ownership; dirty bit is another name for owned bit.

---

The shared bit is set in the cache tag if the data is also present in another processor's cache. The dirty bit is set if the data has been modified by the processor and the changes have not yet been written to memory. The dirty bit will be set only if the processor is operating directly on the MBus.

### ***Consistency with MMU Accesses***

Accesses initiated by the MMU, either reads for table walks or writes for R&M updates, access the bus directly (they never access internal caches). When software accesses page tables in memory, the accesses might be cacheable. This can lead to problems because the MMU will not access the copy cached in the data cache. Software must be sure to access the page tables appropriately. The options for proper operation are different between direct MBus and VBus configurations.

In direct MBus configurations, reads and writes initiated by the MMU do not snoop the data cache. Thus page tables must be accessed through non-cacheable memory space to ensure consistency. There is no external cache, so MCNTL.TC must be deasserted.

The write-through nature of the data cache on VBus keeps the data cache and the external cache consistent. Thus the software may make cacheable access to the page tables, and the MMU will access the entries in external cache if MCNTL.TC is set. Alternatively, software may access the tables as non-cacheable data, and the MMU will access the entries in main memory if MCNTL.TC is clear.

### ***Consistency with Instruction Accesses***

See Subsection 10.2.5.

## 10.5 Data Cache Diagnostic and Control Interfaces

This section describes the low-level diagnostic and control interfaces to the data cache. These interfaces are accessed via LDA and STA instructions to ASIs assigned for this purpose. Table 10-7 shows the ASIs used for diagnostic and control access to the data cache.

Table 10-7. Data Cache ASIs

Function	ASI
Data Cache Flash Clear	0x37
Data Cache Tags	0x0e
Data Cache Data	0x0f

### 10.5.1 Data Cache Flash Clear (ASI=0x37)

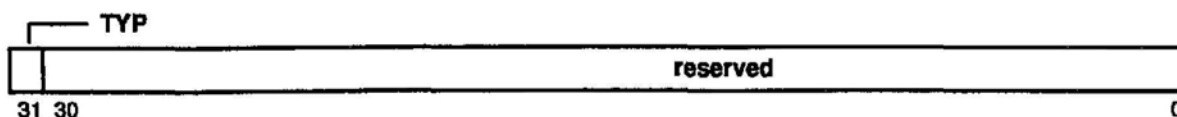
You can invalidate the entire data cache or clear all the lock bits by issuing a store alternate with the ASI value 0x37. The data size of the store operation must be a 32-bit word. All other data sizes cause a `data_access_exception`. The data issued by the store operation is ignored. The most significant bit of the address determines the type of operation.

**Note:**

Flash clear operations should always be executed before enabling the data cache.

The address format is shown in Figure 10-9.

Figure 10–9. Data Cache Flash Clear Address Format



**TYP** Type of operation.

- 0: Invalidate. The valid bit in each of the four PTags of each set in the data cache is cleared. The other fields of the PTag (D, S, and PADDR) are not affected; all USE bits are cleared in the STags of each set in the data cache. The LCK bits of the STags are not affected.
- 1: Unlock. All LCK (lock) bits in the STag of each set in the data cache are cleared. The USE bits of the STag are not affected.

**reserved** Reserved. These bits are ignored on write and read as zero.

### 10.5.2 Data Cache Tags (ASI=0x0e)

Data cache tags are readable and writable with ASI value 0x0e. This direct-access capability is provided for diagnostic purposes.

A PTag is associated with each cache block, and the STag is associated with each set. The PTags and the STags are accessed as 64-bit doublewords. All other data sizes cause a `data_access_exception`. The contents of the cache tags are not affected by watchdog reset or hardware reset.

To access all of the tag information for a block, both the PTag and STag must be accessed. The data cache tag address format is shown in Figure 10–10.

Figure 10–10. Data Cache Tag Address Format

TYP	res	BLOCK	res	SET	res	zero
31 30 29 28 27	26 25		12 11	5 4 3 2	0	

The fields of data cache tag address are:

<b>TYP</b>	Type of operation.
0:	reserved. Generates a data_access_exception.
1:	STag. Accesses the set tag. BLK is ignored. SET selects the set to access.
2:	PTag. Accesses the physical tag. SET and BLK are used to access a particular block's PTag.
3:	reserved. Generates a data_access_exception.
<b>res</b>	Reserved. These bits are ignored.
<b>BLOCK</b>	Block. Selects one of the four blocks (0–3) in a set.
<b>SET</b>	Set selects one of the data cache's 128 sets to access.
<b>zero</b>	Zero field. This field should always be zero.

### PTag Diagnostic Access

Figure 10–11 shows the data cache PTag format.

Figure 10–11. Data Cache PTag Format

res	V	res	D	res	S	res	PADDR
63 57 56 55	49 48 47	41 40 39	24 23				0

The fields of data cache PTag are:

<b>res</b>	Reserved. Read as 0 and ignored on a write.
<b>V</b>	Valid bit for the block. This bit indicates that the block and its associated tag are valid. When this bit is cleared, the corresponding block is invalid, and none of the other fields of the PTag has any meaning. At power-on valid bits are undefined.

- D** Dirty. When the dirty bit is set, it indicates that the block has been modified by one or more writes. A dirty bit can never be set in normal operation in VBus configurations; in the direct MBus configuration, however, it is set whenever a block is modified in the cache.
- In the direct MBus configuration, the data cache is in write-back mode. If the dirty bit is set, the block is copied back into main memory if it must be replaced. See Subsection 10.4.6 for more information.
- S** Shared. When the shared bit is set, it indicates that the block is “shared” between this cache and another cache. Writes to a shared block need special treatment to ensure cache consistency.
- PADDR** Physical address tag. Bits [35:12] of the physical address of the data contained in this cache block.

### STag Diagnostic Access

Figure 10–12 shows the data cache STag format.

Figure 10–12. Data Cache STag Format



The fields of the data cache set tag are:

**res, r** Reserved. Read as 0 and ignored on a write.

**USE**

Used. This bit field indicates which blocks of the set have been accessed recently. The position of the bit in the field determines the block it is associated with. Bits 8–11 correspond to blocks 0–3, respectively. This bit field is updated by the hardware replacement algorithm and is used to select a block for replacement.

USE3	USE2	USE1	USE0
11	10	9	8

**LCK**

Lock bits. Each lock bit can lock a block inside the cache. The position of the bit determines which block it locks. Bit 0 is fixed to 0 and ignores writes. Bits 1–3 lock blocks 1–3, respectively. When a lock bit is set to 1, the corresponding block will not be displaced by the replacement algorithm.

LCK3	LCK2	LCK1	0
3	2	1	0

**10.5.3 Data Cache Data (ASI=0x0f)**

Data cache data is readable and writable with ASI value 0x0f. This direct-access capability is provided for diagnostic purposes. The blocks may only be accessed as 64-bit doublewords. Other data sizes provoke a `data_access_exception`. Data cache data is not affected by watchdog reset, hardware reset, or flash clear. The address format is shown in Figure 10–13.

Figure 10–13. Data Cache Data Address Format

res	BLOCK	res	SET	DWORD	zero
31	27 26 25 24	12 11	5 4	3 2	0

The fields of data cache data address are:

<b>res</b>	Reserved. These bits are ignored.
<b>BLOCK</b>	Block. Selects one of the four blocks (0–3) in a set.
<b>SET</b>	Set. Selects one of the 128 sets of the cache.
<b>DWORD</b>	Doubleword. Selects the doubleword within the cache block.
<b>zero</b>	Zero field. Must be zero.

## 10.6 Store Buffer

The SSP's store buffer is a fully-associative cache of eight doubleword entries. This buffer functions to eliminate most of the performance penalties associated with writes in the write-through cache configurations (VBus) and block replacement in copy-back configurations. Its depth is sufficient to hold 16 SPARC registers, the number required to save a register window to memory.

### 10.6.1 General Operation

A store buffer entry records the state of a pending store operation. The entry contains the address of the store and the data to be stored. Other important information related to the store operation is also recorded in the store buffer. Each store instruction requires a single store buffer entry, whether its size is byte, halfword, word, or doubleword. A copy-back (replacement of a dirty cache block) requires four store buffer entries for the 32 bytes of the cache block.

The store buffer is first-in first-out (FIFO). The order of stores in the buffer is never altered. The oldest store in the buffer must be retired before the next oldest store is presented to the bus.

A store buffer entry is retired from the buffer when it is acknowledged on the bus. In many systems, this will mean that the write has been completed at the main memory and all snooping caches. In some systems, the bus acknowledgement does not indicate completion. In those systems, the SuperSPARC processor relies on the **PEND** signal from the system to correctly implement the memory model. (See Section 8.7.)

The store buffer is a flushing-type buffer. If the doubleword address of a read transaction matches the doubleword address of any write transaction currently in the buffer, the read will wait until that write has completed before continuing. No data is returned from the store buffer to the instruction pipeline. This type of match will force the buffer to flush to memory as quickly as possible.

When the buffer is full, new store operations will cause the pipeline to stall until a single entry in the buffer is available. The buffer is not forced to flush in this situation.

The store buffer turns sequences of memory references within a cache block into burst write operations on the bus (only in VBus configurations). The burst writes will continue as long as the successive writes in the buffer continue to be within the same cache block.

The store buffer components (tags, data, and control) are accessible via ASI spaces 0x30-0x32 for diagnostic purposes.

### 10.6.2 Store Buffer Operation with the MultiCache Controller

The store buffer is primarily used when SuperSPARC is connected to the MXCC, where all store operations are completed immediately into the store buffer. In this configuration, a store operation will never wait for completion unless the buffer is full or disabled or a TLB miss operation occurs. (See Subsection 10.6.4 for a more complete description.) The state of the data cache (hit, miss, or disabled) does not affect store buffer operation when SuperSPARC is used with the MXCC.

### 10.6.3 Store Buffer Operation directly on MBus

When the SuperSPARC processor is connected directly to the MBus (with no MXCC), only non-cacheable stores and copy-back data (dirty cache blocks that are replaced) go into the store buffer. This is primarily due to the copy-back, write-allocate caching policy used on the MBus.

### 10.6.4 Non-Buffered (Synchronous) Operations

When SuperSPARC is used with the MXCC, there are several cases in which stores cannot or should not be buffered. These cases include:

- ☐ Atomic operations,
- ☐ Certain store alternates (STA),
- ☐ MMU R&M updates, and
- ☐ Operations when the store buffer is disabled.

All of these actions block the execution pipeline until they have completed. Since external stores must be performed in order, each of these conditions forces all entries in the store buffer to be written to memory before the synchronous operation begins.

Table 10-8 shows the ASIs that cause the store buffer to flush all pending stores before any store to the ASI-space can proceed. Accesses to other ASIs do not flush the store buffer.



Table 10–8. Synchronous ASIs

Function	ASI
MMU Probe	0x03
MMU Registers	0x04
MMU TLB Diagnostic	0x06
Instruction Cache Tags	0x0c
Instruction Cache Data	0x0d
Data Cache Tags	0x0e
Data Cache Data	0x0f
Store Buffer Tags	0x30
Store Buffer Data	0x31
Store Buffer Control	0x32
Instruction Cache Flash Clear	0x36
Data Cache Flash Clear	0x37
MMU Breakpoint Diagnostic	0x38
BIST Diagnostic	0x39

### 10.6.5 Data Store Errors

When an exception occurs on a buffered write, a `data_store_error` is generated. Unlike most other trap types, when the `data_store_error` is reported, the store instruction that originated the operation has already completed. The trap PC for a `data_store_error` is generally unrelated to the instruction that started the trap. Furthermore, subsequent instructions may also have completed, and the register contents from which the original store data and address were computed may have been lost.

The store buffer contains enough information, however, to perform the store. The trap handler for `data_store_error` can inspect the contents of the store buffer using diagnostic accesses. The contents of the buffer may be used to restart the write that failed.

`Data_store_errors` take priority over exceptions other than reset. In order for a `data_store_error` to be reported, the `PSR.ET` (enable traps) bit must be set, and the `MCNTL.NF` (no-fault) bit must be cleared.

## External Indications

During a store buffer exception, the SuperSPARC processor, when used with the MXCC, will assert **ERROR** for one cycle. When SuperSPARC is directly on the MBus, it will assert **AERR** until **MFSR.SB** is cleared (for example, by a read).

**Note:**

The **ERROR** or **AERR** signal is also used to indicate error mode and cannot be unambiguously interpreted as indicating a store buffer error.

## Trap Handler

When an error occurs, the store buffer is automatically disabled. A store buffer flush is not initiated, and no other writes will be issued from the store buffer. All buffer entries, including the faulting one, are retained in the buffer. All subsequent writes (nominally in the trap handler) are synchronous, bypassing the buffer. This will continue until the store buffer is re-enabled.

Once in the trap handler, the faulting write can be retried. This is accomplished by loading the faulting address and data directly from the store buffer into registers (using the diagnostic ASI access to the buffer's address and data information directly). A store alternate through the MMU pass-through ASIs can then be used to perform an untranslated store to this physical address. In this case, the **MCNTL.AC** (alternate cacheable) bit will be used to determine cacheability. The **AC** bit should be set to the same value as the **C** bit field of the store buffer tag register (since this is the state of the **C** bit in the original transaction).

The data store error handler code should set the **MCNTL.NF** (no-fault) bit before attempting to retry the operation. After the STA to retry the operation, the **MFSR** error bits should be checked explicitly to determine whether the operation was successful (see Subsection 9.12.3.7). If the **MCNTL.NF** bit is not set, an error on the retry of these transactions can cause entry into error mode.

If this retry fails, system software must decide how to continue recovery efforts. In VBus configurations, the error is from a store operation in the current context (since context changes always force a store buffer copy-out, pending stores from another context cannot be present). Once the faulting process is identified, the process can be interrupted or killed, rather than the entire system stopped.

### **Note:**

For SuperSPARC on MBus (with no MXCC), a data store error resulting from a copy-back operation is not guaranteed to be from the current context. In this case, isolating faulting accesses to the process that caused them is more difficult. This situation may still be recoverable, but recovery may be very complex.

### **Memory Order in the Presence of Errors and Recovery**

Standard ordering of memory transactions can be maintained even in the presence of store buffer errors and recovery operations. As long as the recovery routines retry store buffer operations in the order they were requested, no ordering is changed. Any memory operations within the trap handler can be considered to have executed "before" these buffered writes are performed.

### **Recovery by Reenabling the Store Buffer**

When a store buffer exception (`data_store_error`) occurs, the store buffer pointers retain their present value, as if the store were given a retry acknowledgement by the bus. Thus, the trap handler can also re-issue the faulted store by simply re-enabling the store buffer. This works for both cacheable and non-cacheable stores.

Setting `MCNTL.NF` will prevent another store buffer trap from being taken if the store sees another exception. In this case, the store buffer will still turn off, even when `MCNTL.NF` is set. However, the store buffer error remains pending, and the integer unit pipeline does not see the error until `MCNTL.NF` is clear. The `SBCNTL.ER` (error pending) field in the store buffer control register (see Subsection 10.7.3) indicates the presence of a pending (unacknowledged) store buffer error. The trap handler might also check if the store buffer is still enabled after the store is retried.

### **Accessing Store Buffer State for Error Recovery**

When a store buffer exception (`data_store_error`) occurs, the SuperSPARC processor retains information for all pending stores, including the store which encountered the exception, in the store buffer. The following can be accessed in several ASI control spaces:

- ☐ Store buffer control (ASI 0x32),
- ☐ Store buffer tag (ASI 0x30),
- ☐ Store buffer data (ASI 0x31), and
- ☐ MMU.SB (ASI 0x04).

These store buffer entries assist recovery from a store buffer exception. The SBCNTL.DPTR (drain pointer—see Subsection 10.7.3) is left intact after an exception to allow software recovery. In order to re-enable the store buffer after a `data_store_error` was taken, the SBCNTL.DPTR must be initialized to allow proper execution.

#### 10.6.6 Disabled Operation—Strong Ordering

The Store Buffer is disabled when SuperSPARC is first powered up. As a result, all stores are synchronous and block the execution pipeline until they complete. During normal operation, disabling the store buffer allows applications to be run with strong sequential ordering of memory references.

Since disabling the store buffer requires a store to a synchronous ASI, the store buffer will always flush before it can be disabled via store buffer control.

In most system configurations, strong ordering will substantially slow processing speed. See Chapter 8 for more information on Strong Ordering and the other memory models.

## 10.7 Store Buffer Diagnostic and Control Interfaces

This section describes low-level diagnostic and control interfaces to the store buffer. These interfaces are accessed via LDA, LDDA, STA, and STDA instructions to a ASIs assigned for this purpose. Table 10-9 shows the ASIs used for diagnostic and control access to the store buffer.

Table 10-9. Data Cache ASIs

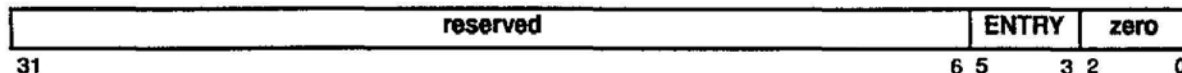
Function	ASI
Store Buffer Tags	0x30
Store Buffer Data	0x31
Store Buffer Control	0x32

### 10.7.1 Store Buffer Tags (ASI= 0x30 )

This ASI performs reads and writes, for diagnostic and error-recovery purposes, to the store buffer's physical tags.

The address format is shown in Figure 10-14.

Figure 10-14. Store Buffer Tag Address Format



The bit fields are:

**reserved** Reserved. These bits are ignored.

**ENTRY** Entry number. This field selects one of the eight entries of the store buffer to access.

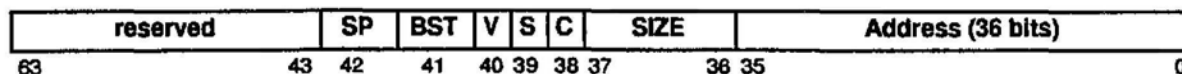
**zero** Field of zeros. Must be zero.

Accessing the store buffer tags with an access size other than a doubleword will result in a data\_access\_exception.

The address selects the tags to access. Tags are read from the selected entry with an LDDA instruction or written to the selected entry with a STDA instruction.

The format of a store buffer tag is shown in Figure 10-15.

Figure 10-15. Store Buffer Tag Format



<b>reserved</b>	These bits are read as 0s and ignored for writes.
<b>SP</b>	Store Barrier Bit. A STBAR instruction sets this bit in the most recent entry in the store buffer, if there are any entries. Newly allocated store buffer entries have this bit clear. Once set, SP remains set in an entry until the entry is retired. See Section 8.7 for more information.
<b>BST</b>	Burst Mode Access. This bit, if set, indicates that the next entry in the store buffer corresponds to the next consecutive address and can thus be issued in burst mode on VBus. This bit is cleared in newly allocated entries.
<b>V</b>	Valid Bit. If set, it indicates that the entry is valid. This bit is cleared when the entry is retired.
<b>S</b>	Supervisor Bit. If set, it indicates that the entry is for a store issued when PSR.S was set and so the entry belongs to a supervisor process.
<b>C</b>	Cacheable Bit. If set, it indicates that the entry is a cacheable access. See Table 10-5 for data cacheability.
<b>SIZE</b>	Size of transaction encoded according to Table 10-10.

Table 10-10. Transaction Table

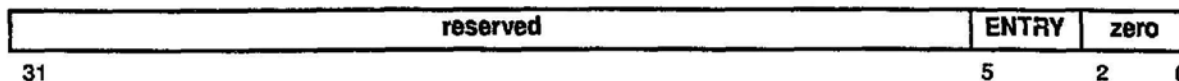
Size	Data Quantity
00	Byte
01	Halfword
10	Word
11	Doubleword

**Address** Address for the Store Buffer Entry. This address must be correctly aligned for the size of the transaction (see Table 10-4). For instance, the lower three bits of the address of a doubleword store buffer entry must be 0; if not, a `memory_address_not_aligned` exception is generated.

### 10.7.2 Store Buffer Data (ASI=0x31)

This ASI is used to perform reads and writes to 64-bit data stored in store buffer entries. Data entries are addressed in the format shown in Figure 10-16, which is the same as the store buffer tag address format.

Figure 10–16. Store Buffer Data Address Format



**reserved** Ignored. Should be zero.

**ENTRY** Entry number. One of the eight entries of the store buffer are selected by the entry field.

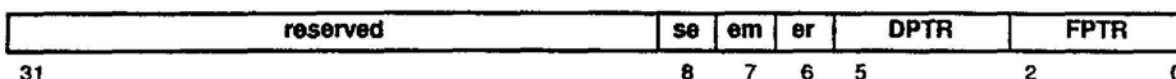
**zero** Zero field. Must be zero.

The store buffer data must be accessed as doublewords. Other data access sizes provoke a `data_access_exception`.

### 10.7.3 Store Buffer Control Register (SBCNTL) (ASI=0x32)

The store buffer control register (SBCNTL) containing the fill and drain pointers is accessible with the ASI value 0x32. SBCNTL must be accessed as a single-word. See Figure 10–17. All other sizes of access will generate a `data_access_exception`. The address field is not used for this access, and any address can be used.

Figure 10–17. Store Buffer Control Register Format



**reserved** These reserved bits read as 0s and are ignored on writes.

**se** Store Buffer Enabled. This bit is read-only and indicates whether the store buffer is enabled (1) or disabled (0). This is a shadow copy of the MCNTLSB bit and is provided for convenience.

**em** Store Buffer Empty. This bit is read-only. SBCNTLEM indicates whether the store buffer is empty (1) or non-empty (0). It is set at reset.

**er** Store Buffer Error Pending. This bit is read-only and indicates whether an untaken store buffer error is pending. This bit is set when a store buffer exception occurs while traps are disabled (PSR.ET=0). This bit is cleared by taking the store buffer exception trap, which occurs automatically when traps are re-enabled. The bit is also cleared on reset.

**DPTR** Drain Pointer. The drain pointer indicates the first entry of the store buffer that would perform a bus transaction.

**FPTR** Fill Pointer. The fill pointer indicates the first entry of the store buffer where a new store request can be written. If it is equal to the drain pointer, the store buffer is full.

The number of pending stores is determined by subtracting the drain pointer from the fill pointer (modulo 8). When the two pointers are equal and there are valid entries, the store buffer is full and cannot accept any more entries. The buffer is empty when the two pointers are equal, but there are no valid entries.





# **Floating-Point Unit Operation**

---

The SuperSPARC Floating Point Unit (FPU) operates in accordance with *The SPARC Architecture Manual*. This chapter details some of the FPU's operations, concentrating on the floating-point-queue interface, special numeric cases, and floating-point exceptions.

<b>Topic</b>	<b>Page</b>
11.1 Floating-Point Instructions .....	11-2
11.2 Floating-Point Deferred Trap Queue (FQ) .....	11-8
11.3 Timing of FPOps .....	11-9

## 11.1 Floating-Point Instructions

SuperSPARC's FPU executes floating-point operate instructions (FPops). It also participates with the integer unit (IU) in executing floating-point events (FPEvs). An FPEv is one of the following:

- ☐ An LDF, STF, LDDF or STDF instruction (load or store floating point registers).
- ☐ An LDFSR or STFSR instruction (load or store the Floating Point Status Register).
- ☐ An STDFQ instruction (stores the Floating Point Deferred Trap Queue).
- ☐ An SMUL, SMULcc, UMUL, UMULcc, SDIV, SDIVcc, UDIV or UDIVcc instruction (these integer multiply and divide instructions execute in the FPU).

FPops include all instructions (such as FMULS) that calculate a result into a floating point (*f*) register from operands in *f* registers. FPops include FBfcc instructions, floating point move (FMOVS) instructions, compare (such as FCMPD) instructions, and convert (such as FSTOD) instructions.

SuperSPARC completes all FPops in program order. Since only a single FPop can be issued to the FPU per instruction group, FPops arrive one at a time at the FPU. In the FPU they are entered in the Floating Point Deferred Trap Queue (FQ). Once the needed operands and resources are available, the FPop at the head of the FQ begins execution.

The SuperSPARC FPU is interlocked for register dependencies. An FPop waits in FQ until its register operands are ready. Usually, a dependency is on a previous FPop that is computing a new value for an *f* register. When the new value has been computed, it will be forwarded directly to the second FPop as it enters execution. Some references call this "chaining."

The following paragraphs detail important aspects of floating-point operation on the SuperSPARC processor (SSP). Many are clarifications or implementation choices from the SPARC Architecture.

### 11.1.1 FBfcc

An FBfcc instruction can immediately follow an FCMP instruction on SuperSPARC. Some previous SPARC FPUs required at least one instruction between the compare and the branch, and the SPARC Architecture specifies that this instruction is required. Programs that do not need to be portable to other SPARC implementations may use the FCMP/FBfcc sequence without any instructions between them.

### 11.1.2 *d* Register Addresses

FPOps and FPEvs that access double-precision floating-point registers (*d* registers) ignore the least significant bit of the register index and access an aligned register pair as if the least significant bit were zero.

The SPARC Architecture recommends an exception for non-aligned *d* and *q* (quad-precision) floating-point register accesses. SuperSPARC does not generate this exception.

### 11.1.3 FSR Version and Implementation Fields

FSR.VER and FSR.IMPL are always zero in the SSP's FPU.

### 11.1.4 Unfinished FPop Exceptions

SuperSPARC completes all FPOps in hardware or generates the appropriate IEEE\_754\_exception. Therefore, unfinish\_FPop exceptions are never generated. Previous FPUs generated unfinish\_FPop exceptions when subnormal operands or results were encountered. SuperSPARC handles these cases in hardware. Extra cycles may be required when an operand or result is subnormal. (See Section 11.3).

### 11.1.5 NaN to Integer Conversion

When the operand to a convert-to-integer instruction (FSTOI or FDTOI) is an NaN (Not A Number), SuperSPARC always produces 0 as the result. Some previous implementations of the FPU produce either 0x80000000 for -NaN or 0x7FFFFFFF for +NaN.

### 11.1.6 NaN Results

When SuperSPARC's FPU produces a result that is an NaN, it is in one of the formats shown in Table 11-1. This is different from earlier SPARC FPU implementations and from the IEEE 754 standard.

When SuperSPARC produces an NaN result, it is stored in one of the fixed formats shown in Table 11-1.

Table 11-1. NaN Output Representation Values

Single Precision:	0x7fc0_0000
Double Precision:	0x7ff8_0000_0000_0000

An NaN can only be generated when NaN reporting is masked by FSR.TEM.NVM = 1.

### **11.1.7 Rounding Operations and Underflow Detection**

SuperSPARC's FPU detects underflow after the rounding operation. The IEEE 754 specification allows detection either before or after rounding. The SPARC Architecture also allows for either method.

### **11.1.8 Quad Precision and Extended Precision**

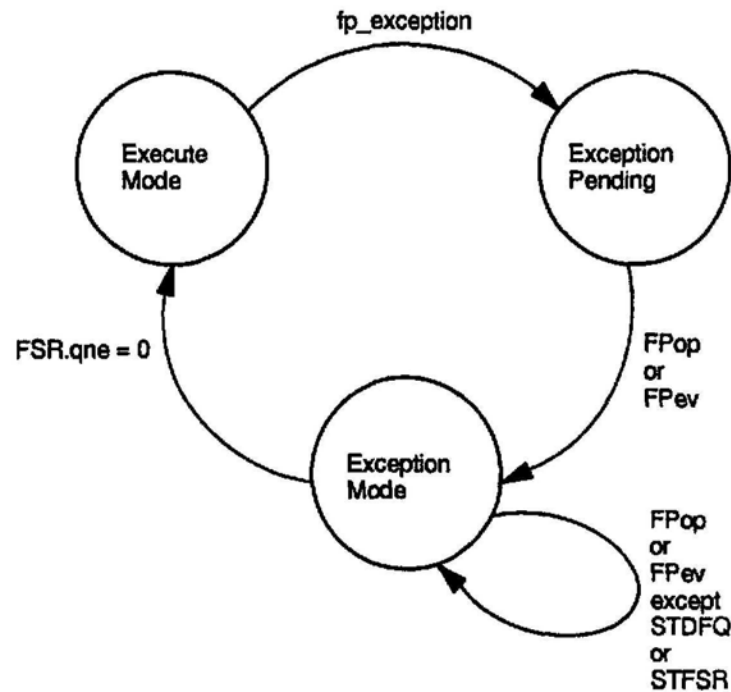
SuperSPARC does not support quad-precision (128-bit) floating-point operations. Any attempt to execute a quad precision FPop will trap with an `floating_point_exception` with `FSR.ftt = 3` (unimplemented FPop). There are no quad-precision FPEvs in *The SPARC Architecture Manual*.

The SPARC Architecture, version 7, did not have quad-precision operations but had extended-precision (80-bit) operations instead. The extended-precision operations of version 7 have been replaced by the quad-precision operations in SPARC, version 8.

### **11.1.9 Floating-Point Exceptions**

SuperSPARC implements deferred floating-point exceptions using the FQ. A floating-point exception will remain pending until another floating-point operation is requested, at which point the pending exception will be reported. Recognition of exceptions are governed by the state machine shown in Figure 11-1.

Figure 11-1. State Machine of FPU States



Once an exception is taken, the `fp_exception` trap handler must empty the FQ before attempting to execute any FPop or FPevs, except for STDFQ and STFSR instructions. An attempt to execute a floating-point instruction after a trap and before the FQ has been emptied will generate an `fp_sequence_error` exception.

If an exception first occurs in the same cycle in which a new floating-point instruction is being sent to the FPU (main pipeline's E0 stage), the exception does not cause a trap on the new floating-point instruction. The exception remains pending until the next floating-point operation is sent to the FPU. If the new floating-point instruction is an FPev (load or store) that is either dependent on an FPop in the queue or modifies an operand to an FPop in the queue, the main pipeline will be held until the queued FPop completes, and any exception it raises will be reported immediately to the waiting FPev.

The presence of processor pipeline hold conditions (particularly from data cache misses) can delay the acceptance of a floating-point exception. The exception will not be taken until a floating-point instruction is in E0 and the pipeline is not being held.

SuperSPARC reports exceptions immediately to instructions that are dependent on the result of an instruction that generates an exception. Otherwise, the exception will be reported to a later floating-point operation.

When an `fp_exception` is signalled, `FSR.ftt` contains the trap type for this floating-point trap. `FSR.ftt` is encoded according to Table 11-2.

Table 11-2. Floating-Point Trap Type (*ftt*) Field of *FSR*

<i>ftt</i>	Trap Type
000	None
001	IEEE_754_exception
†010	unfinished_FPop
011	unimplemented_FPop
100	sequence_error
†101	hardware_error
†110	invalid_fp_register
†111	reserved

† - SuperSPARC does not signal these trap types.

SuperSPARC generates only three of the `fp_exception` types:

IEEE_754_exception	The floating-point exception is an invalid exception, overflow exception, underflow exception, division by zero exception, or inexact exception as encoded in <code>FSR.cexc</code> . <code>FSR.aexc</code> and <code>FSR.fcc</code> are not affected by the instruction that caused this exception.
unimplemented_FPop	An attempt was made to execute an unimplemented (or undefined) FPop.
sequence_error	After an exception was signalled, a floating-point instruction other than <code>STDFQ</code> or <code>STFSR</code> was attempted before the FQ was emptied.  A <code>sequence_error</code> will be reported and recorded in the <code>FSR</code> when a new floating-point request is issued while the FPU is in exception mode. The exception will be reported to the instruction causing the sequence error.

### 11.1.10 Non-Standard Mode

SuperSPARC's FPU does not have a non-standard mode (`FSR.NS`). The `FSR.NR` bit can be read or written but has no effect on floating-point execution.

### **11.1.11 Integer Multiply**

Since integer multiply instructions (i.e., SMUL, SMULcc, UMUL, UMULcc) use the multiplier of the FPU, the timing of integer multiply operations and floating-point operations are interrelated. Integer multiply operations only start if the floating-point queue is empty or if the FPU is in exception mode. Normal floating-point operations will not resume until the integer multiply has completed. Integer multiply operations do not affect the floating-point registers or floating-point condition codes (FSR.fcc).

Integer multiply operations cannot cause any exceptions.

### **11.1.12 Integer Divide**

Since integer divide instructions (SDIV, SDIVcc, UDIV, UDIVcc) use logic in the FPU, the timing of integer divide operations and floating-point operations are interrelated. Integer divide operations start only if the floating-point queue is empty or if the FPU is in exception mode. Floating-point operations will not resume until the integer divide has completed. Integer divide operations do not affect the floating-point registers or floating-point condition codes (FSR.fcc).

Integer divide operations can cause `divide_by_zero` and `illegal_instruction` exceptions but do not cause any floating-point exceptions.



## 11.2 Floating-Point Deferred Trap Queue (FQ)

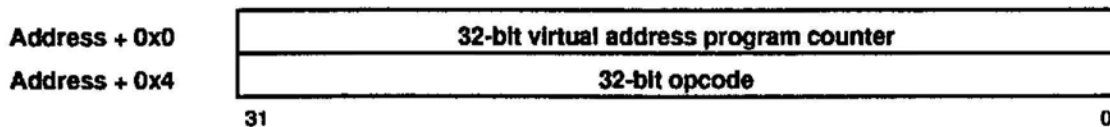
SuperSPARC's FPU contains a queue that holds up to four entries corresponding to FPop's that have been issued to the FPU but have not completed. Each entry has the instruction and the virtual address from which it was fetched. FPevs are not entered into the FQ.

The contents of the FQ can be stored to memory with STDFQ instructions. STDFQ should be issued only when the FPU is in exception mode and FSR.qne = 1. If an STDFQ instruction is executed on a SuperSPARC FPU when not in exception mode, the processor will hold the main (integer) pipeline until all floating-point operations have completed and then store the information from the last completed FPop.

The FQ is not initialized at reset. Therefore, executing STDFQ instructions before any FPop's have executed will store undefined values.

A STDFQ will store a doubleword to memory in the format shown in Figure 11-2. Each doubleword is a single queue entry containing an FPop instruction and the address from which it was fetched.

Figure 11-2. Floating-Point Queue Format



Each time an STDFQ instruction is executed while in exception mode, the next entry in the queue will be stored. The FSR.qne bit should be checked before each store to identify the last valid queue entry.

### 11.3 Timing of FPop

The SSP can start a new FPop every cycle, but the different floating-point operations require different numbers of cycles to complete. Table 11-3, Table 11-4, and Table 11-5 summarize the latency values of the Super-SPARC FPU.

The execution latency of each of these FPop depends on the source(s) and destination data formats. The notation used in these tables designates:

**n** floating-point normal number.

**s** floating-point subnormal number.

**ns=n** Identifies source 1, source 2, and destination data formats, respectively.

*Table 11-3. Floating-Point Operation Latency—Variable-Latency FPop*

INSTRUCTION	nn=n	nn=s	sn=n	sn=s	ns=n	ns=s	ss=n	ss=s
fdivd	9	10	12	13	13	-	13	-
fdivs	6	7	9	10	10	-	10	-
fmuld	3	4	6	7	7	8	8	-
fmulis	3	4	6	7	7	8	8	-
fsqrd	12	-	15	-	-	-	-	-
fsqrs	8	-	11	-	-	-	-	-
fsmuld	3	-	6	-	7	-	8	-
idiv	18	-	-	-	-	-	-	-
idivcc	18	-	-	-	-	-	-	-
imul	4	-	-	-	-	-	-	-
imulcc	4	-	-	-	-	-	-	-

Table 11–4. Floating-Point Operation Latency—Three-Cycle FPOps

INSTRUCTION	nn=n	nn=s	sn=n	sn=s	ns=n	ns=s	ss=n	ss=s
fadds	3	3	3	–	3	–	3	3
fadd	3	3	3	–	3	–	3	3
fsubs	3	3	3	–	3	–	3	3
fsubd	3	3	3	–	3	–	3	3
fcmps†	3	3	3	–	3	–	3	3
fcmpd†	3	3	3	–	3	–	3	3
fcmpes†	3	3	3	–	3	–	3	3
fcmped†	3	3	3	–	3	–	3	3

†Only result is condition code (fcc).

Table 11–5. Floating-Point Operation Latency—Unary Latency FPOps

INSTRUCTION	n=n	n=s	s=n	s=s
fstoi	3	–	3	–
fstod	3	–	3	3
fdtoi	3	–	3	–
fdtos	3	3	3	3
fitos	3	–	–	–
fitod	3	–	–	–
fmovs	3	–	–	3
fabss	3	–	–	3
fnegs	3	–	–	3

# Traps

This chapter describes SPARC traps and the SuperSPARC processor's (SSP's) implementation of them. The traps are implemented to the definition in *The SPARC Architecture Manual*.

Topic	Page
12.1 Introduction .....	12-2
12.2 Trap Types .....	12-4
12.3 Exceptions and Program Counters .....	12-7
12.4 Trap Priorities .....	12-8
12.5 Error Mode .....	12-9
12.6 Traps and the Store Buffer .....	12-10
12.7 Interrupts .....	12-11
12.8 Trap Descriptions .....	12-13

## 12.1 Introduction

A trap is a vectored transfer of control to supervisor software through the trap table. Traps are caused by one or more of the following:

☐ Enabled exceptions.

Exceptions are generated by the execution of a particular instruction. Execution of the instruction generating the exception may be simulated by supervisor software or retried after intervention by supervisor software, or supervisor software may abort the program which generated the exception. Most types of exceptions cannot be disabled; however, some types of fp\_exceptions may be masked (see Section 4.9), and memory and Memory Management Unit (MMU) exceptions can be masked with MCNTL.NF (see Chapter 9).

☐ Errors.

Errors result from serious hardware problems or supervisor software malfunctions that generally cannot be attributed to a single instruction and cannot be recovered. Errors are not maskable.

☐ Resets.

Resets are described in Chapter 13.

☐ Interrupts.

Interrupts are externally or internally generated asynchronous events. Externally generated interrupts are usually caused by I/O events. SuperSPARC's only internally generated interrupts are from the breakpoint facility, as described in Chapter 15.

A trap is always reported to a particular instruction that will be identified as the trapping instruction. Exceptions are always reported to the instruction that caused them, except for floating-point IEEE 754 exceptions, which are deferred and reported to a later floating-point instruction. An error generally cannot be reported to the instruction that caused it, and it is reported to a later instruction. Interrupts are reported to an instruction in the E0 pipeline stage at the time the interrupt is recognized. If possible, error mode reset is reported to the instruction that caused it. Hardware reset acts immediately.

A trap is processed automatically and performs a series of steps. The trap:

- 1) Stops executing the trapping instruction and any instructions after it that may have started execution.
- 2) Completes any pending stores from the store buffer.
- 3) Saves PSR.S in PSR.PS and sets PSR.S.

- 4) Clears PSR.ET.
- 5) Decrements CWP to a new window (such as SAVE), but without checking for window overflow.
- 6) Saves the PC in the new window's local 1 (%l1 or %r17) and nPC in the new windows' local 2 (%l2 or %r18).
- 7) Starts executing from the trap vector location for this trap type.

## 12.2 Trap Types

Each trap has a trap type that indicates the kind of exception, error reset, or interrupt that generated it. For some trap types, additional registers must be examined to determine the precise cause of the trap.

The trap types that occur in the SSP are shown in Table 12-1. The table lists the following for each type of trap:

☐ Trap priority rank.

Describes the exception priority rank. In the event that more than one exception is detected at a given instruction, the trap with the lowest priority rank will be taken.

☐ Type number.

The value to which TBR.tt will be set.

☐ Trap table offset.

The offset to the first instruction of the trap handler.

The trap table has room for four instructions of the trap handler. Longer trap handlers can branch to the remainder of the handler outside the trap table. See Section 4.4.

Table 12-1. Traps Supported by SuperSPARC

TRAP NAME	PRIORITY RANK	TRAP TYPE	TRAP TABLE OFFSET
Reset	1	not set	0x000
data_store_error	2	0x2b	0x2b0
instruction_access_exception	5	0x01	0x010
privileged_instruction	6	0x03	0x030
illegal_instruction	7	0x02	0x020
fp_disabled	8	0x04	0x040
cp_disabled	8	0x24	0x240
window_overflow	9	0x05	0x050
window_underflow	9	0x06	0x060
mem_address_not_aligned	10	0x07	0x070
fp_exception	11	0x08	0x080
data_access_exception	13	0x09	0x090
tag_overflow	14	0x0a	0x0a0
division_by_zero	15	0x2a	0x2a0
trap_instruction	16	0x80 to 0xff	0x800 to 0xff0
interrupt_level_15	17	0x1f	0x1f0
interrupt_level_14	18	0x1e	0x1e0
interrupt_level_13	19	0x1d	0x1d0
interrupt_level_12	20	0x1c	0x1c0
interrupt_level_11	21	0x1b	0x1b0
interrupt_level_10	22	0x1a	0x1a0
interrupt_level_9	23	0x19	0x190
interrupt_level_8	24	0x18	0x180
interrupt_level_7	25	0x17	0x170
interrupt_level_6	26	0x16	0x160
interrupt_level_5	27	0x15	0x150
interrupt_level_4	28	0x14	0x140
interrupt_level_3	29	0x13	0x130
interrupt_level_2	30	0x12	0x120
interrupt_level_1	31	0x11	0x110



### Trap Types Not Implemented

SuperSPARC does not implement certain optional trap types defined by the SPARC Architecture. The trap types not implemented by SuperSPARC are shown in Table 12-2.

Table 12-2. Trap Types Not Implemented by SuperSPARC

Trap Type	Explanation
Instruction_access_error	All instruction access faults are signalled as instruction_access_exceptions.
R_register_access_error	SuperSPARC does not detect any condition that could lead to this error.
Cp_exception	SuperSPARC does not support a coprocessor.
Data_access_error	All data access faults are signalled as data_access_exceptions.
Unimplemented_FLUSH	FLUSH is implemented.
Unimplemented_MUL	The various forms of integer multiply are implemented.  The various forms of integer divide are implemented.  SuperSPARC has no additional trap types beyond those in <i>The SPARC Architecture Manual</i> .
Unimplemented_DIV	
Implementation-Dependent-Exception	

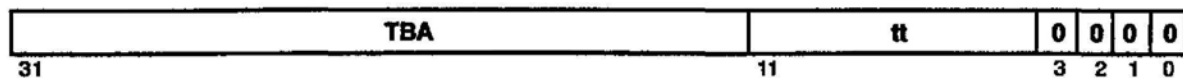
## 12.3 Exceptions and Program Counters

Three program counters are involved when an exception occurs:

- ☐ XPC, the exception program counter.
- ☐ XNPC, the exception next program counter.
- ☐ XHPC, the beginning of the exception handler code.

SuperSPARC recovers the XPC and XNPC values from one of the many program counters maintained in the pipeline, depending on the pipeline stage where the exception occurs. The values are then stored in registers l1 and l2 in the new register window before entry into the trap handler. XHPC is the TBR, which is computed from the TBA register and the trap type (tt), as shown in Figure 12-1.

Figure 12-1. Exception Handler Program Counter (XHPC)



## **12.4 Trap Priorities**

When one or more exceptions, errors, interrupts, or resets occur on a single instruction, only the condition with the lowest priority rank will be recognized and trapped. The priority rank of the trap types in SuperSPARC is shown in Table 12-1.

SuperSPARC can execute more than one instruction in a given cycle, and each of those instructions can have one or more exceptions. All exception requests propagate through the instruction pipeline and are resolved only in the last pipe stage (WB). Only the exception from the earliest (in program order) excepting instruction will be recognized and trapped. Priority ranks are never compared between two instructions. A low-priority exception at an earlier instruction in a given instruction group has precedence over a high-priority exception at a later instruction in the same group.

## 12.5 Error Mode

Error mode can be entered when any exception or error occurs while the ET bit is cleared. In general, these are considered fatal errors, though system software may be able to recover in certain cases.

Supervisor should be careful not to generate exceptions when PSR.ET is clear. A trap handler can set MCNTL.NF immediately on entry. The MCNTL.NF (No-Fault) bit disables reporting of data\_access\_exceptions and other memory errors (such as bus errors) to the CPU.

Error mode generates a watchdog reset trap, which acts like any other trap, with a few differences. For most error-mode traps, the TBR.tt (trap type) field is not set. This is to retain the prior value of TBR.tt to help recovery. There is one case where this rule is not applied: If error mode is entered during the execution of an RETT instruction, the TBR.tt field will be set based on the exact cause of certain exceptions. See Section B.26, Return from Trap Instruction, in *The SPARC Architecture Manual*.

In all cases, the PSR.PS (previous supervisor) bit will not be affected by a watchdog reset.

In all cases, the MCNTL.BT (Boot Mode) bit will be set by a watchdog reset.

For more information on error mode, see Chapter 13.

## **12.6 Traps and the Store Buffer**

When the SuperSPARC CPU takes a trap, it disables further traps by clearing PSR.ET (Enable Traps). This makes the CPU vulnerable; if a synchronous (i.e., non-interrupt) exception occurs while PSR.ET is clear, the CPU enters error mode. Error mode will initiate a watchdog reset.

Several steps are taken to help avoid this second exception and error mode.

- ☐ Before traps are disabled (PSR.ET is still 1) and before the exception handler is fetched, the store buffer completes all pending writes to memory. This prevents any user-level faults caused by these writes from occurring at unexpected or unsafe points within the trap handler.
- ☐ If an exception does occur on this store buffer flush, a `data_store_exception` is taken instead of the pending trap.
- ☐ If the buffered stores complete successfully, the normal trap-handling sequence continues, and PSR.ET is cleared.

See Subsection 10.6.5.

## 12.7 Interrupts

Interrupt\_level\_n traps are triggered by hardware-interrupt requests. The normal source of these requests is from external pin assertions on the IRL[3:0] pins. In addition to these external requests, SuperSPARC can generate internal interrupt requests from several sources.

External interrupts are generated by the system in a system-dependent manner. Their definitions are also system-dependent. SuperSPARC's IRL[3:0] pins are level-sensitive. System hardware can control these pins in an implementation-dependent manner to implement different system-interrupt architectures. When used with MXCC, the IRL pins are driven directly from MXCC. For additional information on interrupts when using MXCC, see Chapter 16.

The IRL[3:0] pins are sampled in successive cycles to debounce transients. The IRL level must remain stable for three cycles before it is recognized as an external interrupt request. These are combined with internally generated requests to form the interrupt request. The IRL[3:0] pins are encoded with a single interrupt level. Internal requests are also presented at particular levels. Only the highest priority interrupt is presented to the pipeline.

The PSR.ET (enable traps) bit must be set for any interrupt to be accepted (including non-maskable interrupts). Requests are ignored as long as PSR.ET is clear.

When PSR.ET is set and there is an interrupt request, the request is compared to the current processor interrupt level (PSR.PIL). If the request is at a higher level than the current PIL, or equal to 15 (non-maskable), it is prioritized with other exceptions. The lowest ranking exception is taken.

A valid instruction must exist at the WB stage of the pipeline before the interrupt will take a trap. Interrupt traps are processed like other traps.

### 12.7.1 Interrupt Latency

Maximum interrupt latency is determined by internal and external factors, such as the maximum length of certain floating-point operations, system memory latency times, and the maximum store buffer depth. In addition, interrupt latency can be increased by the time spent executing with PSR.ET clear. PSR.ET is clear on entry into trap handlers and may be cleared at other times by supervisor software.

Note that interrupts are effectively positioned at the last valid instruction in a particular instruction group. In addition, a valid instruction must be present in order for the interrupt to be registered. If the execution pipeline is not progressing (waiting on cache misses, for example), an interrupt will not be taken. Due to this, maximum interrupt latency for SuperSPARC is highly system-dependent.

An undesirable side effect of flushing the store buffer on trap is a higher maximum interrupt latency. Every interrupt requires a copy-out, which in the worst case involves eight unbuffered writes to memory. This delay may be unacceptable for real-time applications. The copy-out requirement can be avoided, however, by running with the store buffer disabled; in this case, the store buffer is always empty. Fast interrupt response is guaranteed, although overall performance is reduced by the resulting synchronous external stores.

### **12.7.2 Internal Interrupt Sources**

The only internal source for interrupt requests is breakpoints. The breakpoint logic can be programmed to cause interrupts at a software-controlled level on certain events. These may be code-, data-, or counter-generated breakpoints. Several registers, including MDIAG.BKC (breakpoint control), ACTION (action on breakpoint event), and the counter registers determine when these interrupts are generated. The level generated for these interrupts is defined in the ACTION.BCIPL (breakpoint and counter-interrupt level).

See Section 15.2.

## 12.8 Trap Descriptions

Table 12-1 shows the trap types that can occur in SuperSPARC. Each type of trap is described below.

### 12.8.1 Reset Trap

The reset trap does not set the trap type, TBR.tt, except for watchdog reset induced by error mode entered from an RETT instruction (see Section 12.5). It is the highest-priority trap.

SuperSPARC recognizes three types of reset:

- ☐ Hardware Reset.
- ☐ Built-In Self-Test (BIST) Reset.
- ☐ Watchdog Reset.

For a detailed description of reset operation, see chapter 13.

### 12.8.2 Data Store Error Trap

The data store error trap is trap type 0x2b, priority 2.

Data\_store\_error is caused when a store buffer copy-out encounters an external bus error (e.g., parity or ECC problem). For a detailed description, see Subsection 10.6.5.

### 12.8.3 Instruction Access Exception Trap

The instruction access exception trap is trap type 0x01, priority 5.

Instruction\_access\_exception may be caused by many events. The normal source of this error is MMU protection errors for pages that have been swapped out of virtual memory space. Bus errors and code breakpoints can also cause this exception. See Subsection 9.12.3 for a detailed description of error reporting for these traps. Note that the fault address register (FAR) is never updated for these errors.

### 12.8.4 Privileged Instruction Trap

The privileged instruction trap is trap type 0x03, priority 6.

Privileged\_instruction trap is caused when PSR.S is cleared and a privileged instruction is issued. There are many classes of privileged instructions. See *The SPARC Architecture Manual* for a complete list.



### 12.8.5 Illegal Instruction Trap

The illegal instruction trap is trap type 0x02, priority 7.

Illegal\_instruction (sometimes also called unimplemented\_instruction) trap is caused when an instruction with an unassigned opcode or illegal opcode is executed. This trap may also signal instructions that are used in inappropriate ways or with invalid fields or data. For example, SuperSPARC issues an illegal instruction trap when an RETT instruction is executed with the PSR.ET bit set and in several data-dependent cases of the WRPSR instruction. An Integer divide instruction can also generate a data-dependent illegal instruction trap if the dividend has significant bits beyond bit 51.

### 12.8.6 Floating-Point-Disabled Trap

The floating-point-disabled trap is trap type 0x04, priority 8.

FP\_disabled trap is caused when PSR.EF is cleared and any floating-point instruction is issued. This trap is never issued for integer multiply or integer divide.

### 12.8.7 Coprocessor-Disabled Trap

The coprocessor-disabled trap is trap type 0x24, priority 8.

SuperSPARC does not provide a coprocessor interface. PSR.EC (enable coprocessor) bit is zero and non-writable. Any attempt to set the PSR.EC bit will generate an illegal\_instruction trap. Any attempt to execute a coprocessor instruction will generate a cp\_disabled trap. SuperSPARC never generates a cp\_exception trap.

### 12.8.8 Window Overflow Trap

The window overflow trap is trap type 0x05, priority 9.

Window\_overflow trap is caused by a SAVE instruction being executed when the next available register window (CWP-1) is marked invalid in the WIM register. Note that this trap is generated only in response to a SAVE instruction; trapping cannot cause this exception.

### 12.8.9 Window Underflow Trap

The window underflow trap is trap type 0x06, priority 9.

Window\_underflow trap is caused by either a RESTORE or RETT instruction, which would cause the incremented value of CWP to point to an invalid register window. In the case of RETT, this trap will immediately lead to error mode and watchdog reset.

### 12.8.10 Memory Address Not Aligned Trap

The memory address not aligned trap is trap type 0x07, priority 10.

Mem\_address\_not\_aligned trap is caused by a load, store, or control transfer operation that violates the correct SPARC memory alignment. The correct alignment of an instruction is a word. Half-word references require the low-order address bit to be zero. Word references require the two address bits to be zero. Double-word references require the three low-order address bits to be zero.

### 12.8.11 Floating-Point Exception Trap

The floating-point exception trap is trap type 0x08, priority 11.

Fp\_exception traps are caused by certain floating-point arithmetic conditions being detected. These exceptions are deferred traps and will only be reported on execution of another floating-point operation (or floating-point event) at some later time. The time taken to cause the exception is variable, depending on the pipeline state, numeric conditions, and the exact sequence of instructions. Improper floating-point error recovery can also cause these exceptions, particularly sequence errors.

The cause of the fp\_exception can be determined from the FSR.ftt field. FSR.ftt can be decoded according to Table 4-3. The instruction that caused the exception can be determined by reading the state of the floating-point queue, described in Section 11.2. (See also Section 4.9.)

### 12.8.12 Data Access Exception Trap

The data access exception trap is trap type 0x09, priority 13.

Data\_access\_exception traps are generally caused by MMU protection errors for load and store operations. They may also be caused by bus errors and data breakpoints. See Section 9.12.3 for a detailed description of error reporting for these exceptions.

### 12.8.13 Tagged Operation Overflow Trap

The tagged operation overflow trap is trap type 0x0a, priority 14.

Tag\_overflow traps are caused by execution of tagged add or subtract and trap on overflow (TADDccTV and TSUBccTV) instructions. The trap will be signalled when the least significant two bits of either source operand is non-zero, or when the operation produces a result that causes the overflow flag to be set.

#### **12.8.14 Integer Divide by Zero Trap**

The integer divide by zero trap is trap type 0x2a, priority 15.

Division\_by\_zero trap is caused when an integer divide instruction is issued with the value of the second operand (denominator) being zero.

If both divide by zero and the illegal instruction conditions exist, the illegal instruction trap will be signalled.

#### **12.8.15 Trap Instructions (Ticc)**

Trap instructions are trap type 0x80-0xff (instruction dependent), priority 16.

The trap type is computed by taking the low-order 7 bits of the sum of the two operands to the Ticc instruction and adding it to 0x80.

Trap\_instruction traps are software-initiated traps caused by the execution of Ticc instructions.

#### **12.8.16 Interrupt Levels**

Interrupt levels are trap type 0x11-0x1f (request level dependent), priorities 17-31.

Interrupt\_level\_n traps are generated by hardware-interrupt requests. The normal source of these requests is from external pin assertions on the IRL[3-0] pins. In addition to these external requests, SuperSPARC can generate internal interrupt requests. See Section 12.7.

Generation and definition of interrupt requests is system-dependent.

# Reset

---

The SuperSPARC processor (SSP) can be reset in three ways:

- ☐ Via its RESET pin,
- ☐ Automatically at the completion of Built-In Self Test, or
- ☐ Automatically after entering error mode.

SuperSPARC must be reset when power is first applied. Its internal state is greatly altered by a reset.

The MultiCache Controller (MXCC) may also be reset in several ways, all of which greatly alter its internal state.

Topic	Page
13.1 Reset Types .....	13-2
13.2 Hardware Reset .....	13-5
13.3 Watchdog Reset .....	13-6
13.4 Built-In Self-Test (BIST) .....	13-7
13.5 Reset in MXCC .....	13-10

### 13.1 Reset Types

The SSP can be reset in three ways:

- ☐ Hardware reset is initiated from outside the processor by asserting the **RESET** signal. SuperSPARC must be reset when power is first applied. It is also possible to cause a hardware reset in emulation mode or by boundary scan.
- ☐ Built-In Self-Test (BIST) generates a second type of reset when it completes. BIST operations can be requested either by software (with a STA) or via the JTAG interface. When BIST is initiated through software using STA, an internal reset is automatically generated. When BIST is initiated by JTAG, however, reset is not automatically generated. This can be done by entering the TAP reset state by either assertion of TMS for five consecutive TCK cycles or asserting TRST.
- ☐ Watchdog reset is an internally generated reset caused by entry into error mode.

The state changes in the SSP due to each type of reset is shown in Table 13-1.

Table 13-1. Register State After Reset

REGISTER	STATE AFTER RESET		
	HARDWARE	BIST	WATCHDOG
Floating Point Queue	Invalidated	Invalidated	Unchanged
Boot Mode (MCNTL.BT)	1 (Boot mode)	1 (Boot mode)	1 (Boot mode)
MMU Enable (MCNTLEN)	0 (MMU disabled)	0 (MMU disabled)	Unchanged
No Fault (MCNTL.NF)	0 (Faults enabled)	0 (Faults enabled)	Unchanged
Data Cache Enable (MCNTL.DE)	0 (Data cache disabled)	0 (Data cache disabled)	Unchanged
Instruction Cache Enable (MCNTL.IE)	0 (Instruction Cache Disabled)	0 (Instruction Cache Disabled)	Unchanged
Store Buffer Enable (MCNTL.SB)	0 (Store Buffer Disabled)	0 (Store Buffer Disabled)	Unchanged
MBus Mode (MCNTL.MB)	1/0 Depends on CCMODE pin	1/0 Depends on CCMODE pin	1/0 Depends on CCMODE pin
Parity Enable (MCNTL.PE)	0 (Parity Disabled)	0 (Parity Disabled)	Unchanged
Snoop Enable (MCNTL.SE)	0 (Snooping Disabled)	0 (Snooping Disabled)	Unchanged
Partial Store Ordering (MCNTL.PSO)	0 (TSO/Strong Ordering)	0 (TSO/Strong Ordering)	Unchanged

Table 13–1. Register State After Reset (Continued)

REGISTER	STATE AFTER RESET		
	HARDWARE	BIST	WATCHDOG
Alternate Cacheable (MCNTL.AC)	0 (Non-cacheable)	0 (Non-cacheable)	Unchanged
Table Walk Cacheable (MCNTL.TC)	0 (Non-cacheable)	0 (Non-cacheable)	Unchanged
Error Mode (MFSR.EM)	0 (Not an error mode, or watchdog reset)	0 (Not an error mode, or watchdog reset)	1 (A watchdog reset)
TLB Lock Bits	0 (All TLB lock bits cleared)	0 (All TLB lock bits cleared)	0 (All TLB lock bits cleared)
Multiple Instruction Mode (ACTION.MIX)	0 (Single Instruction Execution)	0 (Single Instruction Execution)	0 (Single Instruction Execution)
Breakpoints (MDIAG)	0 (All breakpoints disabled)	0 (All breakpoints disabled)	0 (All breakpoints disabled)
Program Counter	0 (PC = 0x0, nPC = 0x4)	0 (PC = 0x0, nPC = 0x4)	0 (PC = 0x0, nPC = 0x4)
BIST Status	00 (No BIST since reset)	01 or 10 (BIST run since reset)	Not Affected
Store Buffer Tags	Valid bits cleared	Valid bits cleared	Valid bits cleared
Store Buffer Contents	Contents Uninitialized	Contents Uninitialized	Contents Unchanged
Data Cache	Contents Uninitialized	Contents Uninitialized	Contents Unchanged
Instruction Cache	Contents Uninitialized	Contents Uninitialized	Contents Unchanged
Register File	Contents Uninitialized	Contents Uninitialized	Contents Unchanged
Processor Status Register (PSR)	S=1, ET=0, EC=0, Ver=0, Impl=4, PSR.CWP Uninitialized	S=1, ET=0, EC=0, Ver=0, Impl=4, PSR.CWP Uninitialized	S=1, ET=0, EC=0, Ver=0, Impl=4, PSR.CWP Unchanged
Window Invalid Mask (WIM)	Uninitialized	Uninitialized	Unchanged
Fault Status Register (MFSR)	Uninitialized (except for MFSR.EM bit)	Uninitialized (except for MFSR.EM bit)	Unchanged (except for MFSR.EM bit)
Shadow Fault Status Register (MSFSR)	Uninitialized	Uninitialized	Unchanged
Emulation Facilities	Disabled	Disabled	Unchanged

**Notes:**

Values shown as “uninitialized” are not set to any guaranteed state after reset. These values should be initialized before use.

Values shown as “unchanged” are not affected by the indicated type of reset.

**Note:**

After any reset, SuperSPARC will execute in single-instruction execution mode. Multiple-instruction-per-cycle execution is enabled by setting the ACTION.MIX bit.

### 13.1.1 Boot Mode

Boot mode is a special mode of the Memory Management Unit (MMU) in which all instruction accesses (and alternate spaces references through ASIs 0x08 and 0x09) pass their virtual address in physical address bits 27 through 0, and the upper eight physical address bits (35 through 28) are set to 0xFF.

Boot mode is entered after any reset (either hardware or watchdog). It may be disabled by clearing the MCNTL.BT bit explicitly. Note that boot mode overrides the MMU enable for instruction accesses. Boot mode does not affect data references.

During boot mode, the MCNTL.AC bit is ignored for instruction references and alternate space transactions through instruction space ASIs (0x08,0x09), making these accesses to instruction space non-cacheable when BT=1. However, the AC bit is still used for data references in this mode.

### 13.1.2 Determining Reset Type

Since all three types of reset take a reset trap with MCNTL.BT set, all enter the reset handler at physical location 0xFF000000. The reset handler should examine MFSR.EM and BIST.STATUS to determine the type of reset. See Table 13-2.

Table 13-2. Decoding Reset Types

REGISTER FIELDS	STATE AFTER RESET		
	HARDWARE	BIST	WATCHDOG
MFSR.EM (Error Mode)	0	0	1
BIST.STATUS (BIST completion status)	00	01 or 10	Unchanged

## 13.2 Hardware Reset

Hardware reset is requested by asserting the **RESET** pin for a minimum of eight cycles or greater and then deasserting it. As soon as **RESET** is asserted, SuperSPARC deasserts or disables all outputs except for TDO and ESB. External logic should monitor **RESET** for the validity of control signals.

### 13.2.1 Hardware Reset Requirements

When SuperSPARC is first powered on, the phase locked loop may need a long period to stabilize. **RESET** must remain asserted during this time. Approximately 100 milliseconds will be required. Unpredictable operation will result if the **RESET** is released before the PLL has stabilized.

At power-on, **RESET** must be held asserted for 16 cycles after the PLL has stabilized. At other times, when both power and the PLL remain stable, **RESET** can be asserted for as few as eight cycles.

---

**Note:**

In order for SuperSPARC to properly reset, care must be taken in the system implementation. In particular, JTAG operation may affect SuperSPARC's ability to reset (the JTAG TAP controller should be in the reset state when hardware reset is asserted).

---

### 13.2.2 Hardware Reset Sequence

SuperSPARC takes several actions in response to a hardware reset request.

Once **RESET** has been deasserted, SuperSPARC spends several hundred cycles initializing internal logic. This initialization takes over 340 cycles and is internally timed and the same for all devices of the same stepping. During this time, the cache column redundancy repair circuits are configured, and all chip outputs are held inactive or under high impedance.

Next, internal registers are set, per table Table 13-1. Then SuperSPARC takes a reset trap (See Chapter 12).

This forces execution to begin at virtual address 0x0. Since boot mode is set, physical address 0xFF000000 is used to fetch instructions from memory. System software may distinguish hardware reset from watch-dog reset by the MFSR.EM (error mode) bit being cleared. Since the ACTION.MIX (multiple instruction execution) bit is cleared, SuperSPARC's superscalar execution is disabled, and a maximum of one instruction may be executed in each cycle.

None of the entries in the store buffer, data cache, instruction cache, or TLB (except lock bits) change. Software should initialize each resource before enabling it.



### 13.3 Watchdog Reset

In addition to the hardware reset, there is an internally-generated reset referred to as a watchdog reset. This reset is caused by entry into error mode. Error mode is described in Section 12.5 and in *The SPARC Architecture Manual*.

To allow recovery from many error-mode conditions, very little state is affected by watchdog reset. The only MCNTL bit affected by a watchdog reset is the MCNTL.BT (boot mode) bit. The MFSR.EM (error mode) bit is set to indicate that this is a watchdog reset, as opposed to a hardware reset. Since the cache redundancy logic has already been programmed (during hardware reset), it is not done again. Breakpoints are cleared at watchdog reset.

In VBus Configurations, SuperSPARC asserts **ERROR** for one cycle, allowing the MXCC (or external system logic) to record the occurrence of error mode. At the completion of this bus cycle, a watchdog reset is generated.

When the SuperSPARC processor is used directly on the MBus without the MXCC, **AERR** is asserted and a watchdog reset is generated. **AERR** remains asserted until MFSR.EM is cleared.

Once the above actions have been completed, a reset trap will be taken and control will pass to physical address 0xFF000000 as with hardware reset.

## **13.4 Built-In Self-Test (BIST)**

SuperSPARC has BIST logic on-chip. BIST is a quick check for device integrity; it is not an exhaustive proof of device function. Many types of device faults will be detected by an incorrect signature value after a BIST. There are two types of BIST: short and long versions. The long BIST operation, though a more exhaustive check of the logic than the short BIST, is not supported.

This section describes how BIST operates, how to initiate BIST, and how to use the results. Also included are warnings regarding BIST operation.

### **13.4.1 Internal BIST Operation**

BIST uses internal logic scan paths to write in pseudo-random test patterns into the chip logic (internal states). One cycle of execution is then run to let the states assume their next states; then state of the logic is captured through the scan path into a signature analyzer. The signature analyzer creates a signature value based on the results from the logic and stores this value in the BIST.SIGNATURE register.

### **13.4.2 BIST Coverage**

The BIST sequence checks all normally scannable logic but does not check the internal memory arrays. The TLB, store buffer, prefetch buffers, cache arrays, and register files are not checked by BIST.

### **13.4.3 Signature**

The correct signature value is known but is device-stepping-dependent. Correct signature values for the device will be published in the data sheet.

Different signature values are generated for the long (unsupported) and short BIST operations.

### **13.4.4 Initiating BIST**

To initiate BIST, a STA to ASI 0x39 is issued. A store to address 0 selects a short BIST. An unsupported long BIST is selected by writing to virtual address 0x100. An ASI 0x39 access to any address other than 0x100 or 0x0 will generate a `data_access_exception`.

Once requested, internal logic controls the BIST operation. An external reset aborts the BIST operation. When the sequence completes, an internal reset is generated (see the hardware reset description above).

BIST may also be initiated through the JTAG interface. For details on JTAG-initiated BIST, see Chapter 21.

### 13.4.5 BIST ASI Operation ASI=0x39 - BIST Diagnostic Interface

There are two memory-mapped MMU-resident diagnostic registers used to support BIST (see Table 13–3 and Table 13–4). Any byte, halfword, and doubleword or swap access into any of these diagnostic registers is explicitly illegal and will generate a `data_access_exception`. LDA/STA single only is allowed.

Table 13–3. BIST Diagnostic Registers Within ASI 0x39

Address	Load/Store	Data Format	Description
0x00000000	Store	Don't Care (should be zero)	Start Short BIST
0x00000100	Store	Don't Care (should be zero)	Start Long BIST (unsupported)
0x00000000	Load	Signature[31:0]	Read 31-bit signature
0x00000100	Load	Status[31:0](see value below)	Read 2-bit BIST Status

The possible values of the status register are shown in Table 13–4.

Table 13–4. BIST Status Register Values

Value	Meaning
0x00000000	No BIST run
0x00000001	Short BIST complete
0x00000002	Long BIST complete (unsupported)

BIST.STATUS is not affected by watchdog reset, but is cleared to zero by hardware reset.

### **13.4.6 Warnings Regarding BIST Operation**

After BIST finishes, SuperSPARC generates an internal reset. This internal reset behaves similarly to the hardware reset. One of the consequences of this reset is that MCNTL is initialized. In particular, the MCNTL.PE bit is cleared, so that SuperSPARC does not check parity but generates inverted parity on the pins. This internally generated reset is not seen by a cache controller, such as the MXCC, so that if the parity were enabled in the cache controller before starting BIST, it would still be enabled in the cache controller after BIST. This results in parity mismatch between SuperSPARC and the cache controller. Software should check the MXCCCR.PE bit after BIST has completed and adjust the SuperSPARC's MCNTL.PE bit before attempting any writes. To be safe, the following algorithm could be used:

- 1) Before starting BIST, flush the store buffer (even though STA 0x39 also does it).
- 2) Check the MXCC status register and wait until nothing is pending.
- 3) Start BIST.
- 4) After BIST finishes, read BIST status.
- 5) Read BIST signature.
- 6) Read MXCC control register.
- 7) Set SuperSPARC's MCNTL.PE bit if CCCR.PE bit is set.
- 8) Continue.

Even though the MXCC was mentioned as an example, any other system component could have a similar problem since they do not see SuperSPARC's internally generated reset. Thus care has to be taken in recovering the system after BIST finishes.

### 13.5 Reset In MXCC

After a system reset (**RSTIN**), the content of the external cache is not defined, and the external cache is disabled. The external cache is enabled by the start-up software by setting the cache enable (**CE**) bit in the cache controller configuration register. Prior to enabling the cache, however, the software should initialize the cache by clearing the Valid (**V**) and Pending (**P**) bits for each sub-block in the tag for each of the blocks in the cache.

Reset can come from:

- ☐ the system **RSTIN**,
- ☐ **MXCC BIST**, or
- ☐ the **SSP**.

The processor can initiate two different resets:

- ☐ watchdog (**WD**) reset, and
- ☐ software internal (**SI**) reset.

Remote processors on the system bus can initiate only software-internal resets. The reset register is used to determine the type of reset.

#### 13.5.1 System Reset

On system reset, **MXCC** does the following:

- ☐ Asynchronously disables all **DATA/ADDR** output drivers on the **VBus** and all bi-directional output drives on the **MBus/XBus** (all go to high impedance).
- ☐ Drives all control strobes on the **VBus** to high.
- ☐ Resets the SuperSPARC processor by asserting **RESET**.
- ☐ Disables the external cache.
- ☐ Resets all finite state machines.
- ☐ Resets all internal queues.
- ☐ Resets the **MXCC** control register, status register, interrupt pending register, and reset register.
- ☐ Sets the interrupt mask register to 1's.

After system reset, MXCC does the following:

- ☐ Continues to reset the processor for eight cycles.
- ☐ Configures external cache tag RAM column redundancy for 150 cycles. During this time, bi-directional control strobes are three-stated and uni-directional output control strobes are deasserted.
- ☐ After configuring external cache tag RAM redundancy, asserts  $\overline{\text{RGRT}}$  and  $\overline{\text{WGRT}}$ .

### 13.5.2 Software Internal Reset

On software internal reset, MXCC does the following:

- ☐ Deasserts  $\overline{\text{RGRT}}$  and  $\overline{\text{WGRT}}$ .
- ☐ Waits for pending operations to complete (but external cache updates will not be completed).
- ☐ Clears SXP in the status register and the WD bit in the reset register.
- ☐ Resets the processor for eight cycles.

On a software internal reset, the PE bit in MXCC and SuperSPARC may be different. The system software must ensure that both PE bits are identical before issuing the first write after SI.

### 13.5.3 Watchdog Reset

On watchdog reset, MXCC does the following:

- ☐ In MBus configuration, asserts  $\overline{\text{AERR}}$ .
- ☐ Sets the WD bit in the reset register.

### 13.5.4 BIST Reset

At the completion of a built-in self-test (BIST), the MXCC:

- 1) Resets its internal state,
- 2) Clears WD and SI in the reset register,
- 3) Asserts both  $\overline{\text{RGRT}}$  and  $\overline{\text{WGRT}}$ , and
- 4) Resumes normal operation.

### 13.5.5 Reset Register

MXCC's reset register is shown in Figure 16-13. Its contents can be used to distinguish between types of reset.

Table 13-5 shows the values in the WD and SI fields after Hardware, WD, BIST, and SI Resets. The value X in the table is interpreted as "don't care".

Table 13-5. The WD and SI Fields after Reset

Reset Type	WD	SI
Hardware	0	0
Watchdog	1	X
BIST	X	X
Software	0	1

### 13.5.6 MultiCache Controller Reset Requirements

To ensure the proper operation of the MXCC, the following requirements must be met by the system for reset:

- ☐ When MXCC is first powered on, the phase locked loops may need a long period to stabilize. RSTIN must remain asserted during this time. Approximately 100 milliseconds will be required. Unpredictable operation will result if the RSTIN is released before the PLLs have stabilized.
- ☐ At power-on, RSTIN must be held asserted for 16 cycles after the PLLs have stabilized. At other times, when both power and the PLLs remain stable, RSTIN can be asserted for as few as eight cycles.
- ☐ RSTIN can be asynchronous to either or both of BCLK and PCLK.
- ☐ JTAG reset (TRST) must be asserted at power-on for a minimum of 50 ns. TRST can be asynchronous to any or all of BCLK, PCLK, and TCK. Two TCLKs elapse after TRST is deasserted before TMS may be asserted.
- ☐ After RSTIN is deasserted, there should be no requests from XBus or MBus for a minimum of 150 PCLK cycles in order to allow the external cache tag memory column redundancy programming to complete. There also should be no JTAG operations during this time.
- ☐ All the three-state outputs on the MBus or the XBus (as selected by MBSEL) will be placed in their high-impedance state. It is the responsibility of the system logic to assure that these signals remain in their appropriate states with pull-ups as necessary.
- ☐ After a boundary/internal scan test, the TRST and RESET should be asserted in the same way as during power-on reset for the chip to enter normal operation mode.
- ☐ RSTIN should be held deasserted during internal scan.

- ☐ **RESET** is asserted to the processor asynchronously as soon as **RSTIN** is asserted. MXCC keeps asserting **RESET** for eight cycles after **RSTIN** is deasserted. SuperSPARC disables all bi-directional signals on VBus asynchronously when **RESET** is asserted. While **RSTIN** is asserted, MXCC drives the bi-directional VBus signals with weak drivers toward  $V_{CC}$ . After **RSTIN** is deasserted, MXCC drives all the bidirectional control signals to logic high and then releases them before **RESET** is deasserted.





## **Startup Procedure**

---

---

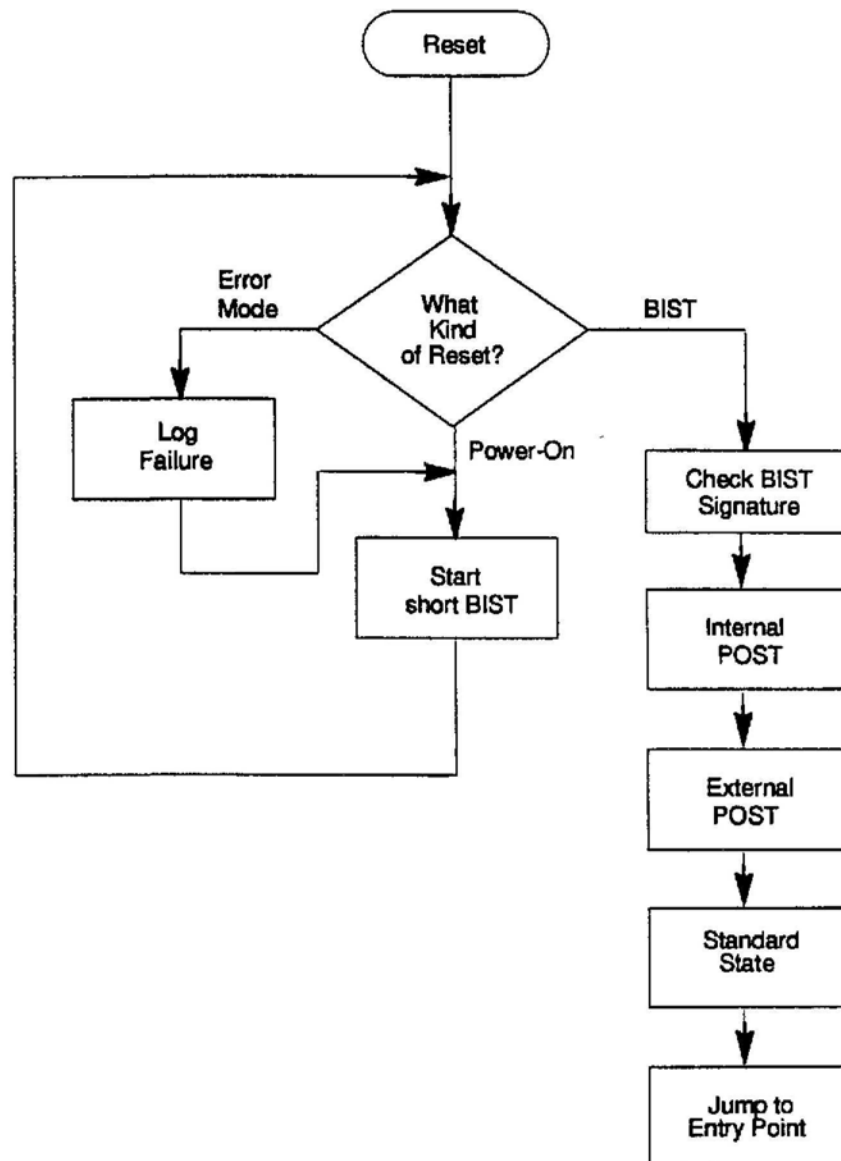
This chapter contains guidelines for developing a reset handler for the Super-SPARC processor (SSP) and the MultiCache Controller (MXCC). An example reset handler is presented.

<b>Topic</b>	<b>Page</b>
<b>14.1 Reset Handling .....</b>	<b>14-2</b>
<b>14.2 Power-On Self-Test .....</b>	<b>14-8</b>
<b>14.3 Reset States .....</b>	<b>14-21</b>

## 14.1 Reset Handling

This is a guideline for writing a reset handler for the SSP. Many aspects of the handler are system-dependent; each system requires the careful development of its own specific reset handler. Figure 14-1 is a flowchart of a reset handler that might serve for a variety of systems. The example reset handler developed in this chapter follows Figure 14-1.

Figure 14-1. Reset Handler Flowchart



In Figure 14-1, power-on reset automatically invokes a short built-in self-test (BIST). When BIST completes, it generates a BIST reset, which completes the initialization of the machine.

See Chapter 13 and Chapter 23 for specific requirements for the timing of **RESET** and **CLK**.

### **Type Determination**

The first step for a reset handler is to determine what kind of reset caused the reset trap. Resets that can cause reset traps are in one of the following three categories (see Section 13.1):

- ☐ Error Mode (Watchdog) Reset,
- ☐ BIST Reset, or
- ☐ Power-on (Hardware) Reset.

Table 13-2 shows the differences between the three reset types that can be used to determine the type of reset.

Error mode resets cause **MFSR.EM** to be set on entry to the reset handler (see Subsection 9.12.1). For other types of reset, this bit is 0. BIST resets set **BIST.STATUS** to non-zero. Hardware reset can be identified because neither **MFSR.EM** nor **BIST.STATUS** is non-zero. This reset handler routine preserves all of the registers except **%y** and **EM\_RST\_SCRATCH** so that the error mode reset handler can log them for debugging. **EM\_RST\_SCRATCH** is a register that can be used by the error mode reset code as scratch (usually an easily reconstructible global register is chosen for this purpose).

On entry to the reset handler (see Table 12-1):

- ☐ **PSR.S** is set (supervisor mode).
- ☐ **PSR.ET** is clear (traps disabled).
- ☐ The Memory Management Unit (MMU) is disabled.
- ☐ **MCNTLBT** (boot mode) is set.
- ☐ The caches are disabled.

Example 14-1 shows the entry and reset type determination for the example reset handler.

**Example 14–1. Reset Handler Entry and Reset Type Determination**

```

reset_handler:
    mov     %g1,%y                ! save %g1 in %y
                                   ! Only %y and
                                   ! EM_RST_SCRATCH are
                                   ! corrupted on the way
                                   ! to em_reset.
                                   ! check_if_error_mode:
    set     0x300,%g1             ! MFSR_RW_VADDR
                                   ! ld & clr mfsr
    lda     [%g1] 0x04, EM_RST_SCRATCH ! MFSR_RW_ASI
                                   ! restore mfsr
    sta     EM_RST_SCRATCH,[%g1] 0x04 ! MFSR_RW_ASI
    set     0x00020000,%g1        ! MFSR_EM,%g1
    and     %g1,EM_RST_SCRATCH,EM_RST_SCRATCH
    mov     %psr,%g1              ! save %psr in %g1
    cmp     EM_RST_SCRATCH,0       ! MFSR.EM=1 ?
    bne     error_mode_reset       ! Yes, error mode reset
    nop                                  ! don't restore psr
                                   ! with unknown

check_if_BIST:
    set     0x00000100,%g1        ! BIST_STATUS_VADDR
    lda     [%g1] 0x39,%g2        ! BIST_STATUS_ASI
    cmp     %g2,0                 ! BIST_STATUS=0 ?
    be      power_on_reset        ! Yes, power-on reset
    nop
    ba,a    bist_reset            ! else BIST reset

```

**14.1.1 Error Mode (Watchdog) Reset**

Error mode reset has occurred if MFSR.EM was set on entry to the handler. On entry, the “borrowed” registers are restored—only the contents of %y and EM\_RST\_SCRATCH have been destroyed. The actions of the error mode reset handler are system-specific. This routine can be used to log the trap type and other registers for debugging. After logging, the routine may attempt to restart the system by treating this as a power-on reset.

Example 14–2 illustrates the routine for the error mode reset.

**Example 14–2. Error Mode Reset**

```
error_mode_reset:
    mov    %g1,%psr          ! restore %psr
    mov    %y,%g1            ! restore %g1
                                ! code goes here to log
                                ! the error mode and
                                ! save registers for
                                ! debugging.
    ba,a    power_on_reset    ! Continue with full
                                ! power-on reset.
```

**14.1.2 BIST Reset**

BIST reset occurs at the completion of a long or short BIST. In the scheme used in Example 14–3:

- 1) Power-on reset initiates a short BIST.
- 2) When the BIST completes, it causes another reset trap.
- 3) After this second type of reset is decoded as a BIST reset, control is passed to BIST\_RESET, where the BIST signature is checked (see Section 13.4).
- 4) The processor is then initialized as if for a power-on reset without BIST.
- 5) Control passes out of the reset handler into the initial program, which might perform additional processor and system tests or load a program.

**Example 14–3. BIST Reset**

```

bist_data:
    .word    BIST_SHORT_SIG
    .word    BIST_LONG_SIG
bist_reset:
                                ! %g2 contains the BIST
                                ! status if 1 was short
                                ! BIST, if 2 was long
                                ! BIST, don't get here
                                ! with %g2=0
                                ! read the BIST
                                ! signature
                                ! BIST_SIG_VADDR
                                ! BIST_SIG_ASI

    lda      [%g0] 0x39,%g3
                                ! BIST_STATUS=2 is long
                                ! BIST

    set      bist_data,%g1
    cmp      %g2,2

    be,a     long_bist_done
    add      %g1,0x4,%g1
                                ! bump pointer if long
                                ! BIST

long_bist_done:
short_bist_done:
    ld       [%g1],%g4
                                ! load correct
                                ! signature
                                ! check actual vs
                                ! correct
                                ! if good, setup normal
                                ! env

    cmp      %g3,%g4
    be       internal_post

    nop
bist_bad_sig:
                                ! If you get here the
                                ! BIST signature
                                ! doesn't match.
                                ! Probably want to do
                                ! some failure logging.

```

**14.1.3 Power-On (Hardware) Reset**

At power-on, some rudimentary internal and external tests are recommended. One test is to run BIST. The routine in Example 14–4 either only starts a short BIST or branches to internal power-on self-test. BIST never returns, but it generates a reset trap at completion.

*Example 14-4. Power-On Reset*

```
power_on_reset:
#ifdef SKIP_BIST_AT_POWER_ON

                                ! here to start a short
                                ! BIST
                                ! BIST_START_SHORT_VADDR
                                ! is 0
                                ! BIST_START_SHORT_ASI

        sta      %g0, [%g0] 0x39
        nop
        nop
        nop

bist_not_started:

                                ! If you get here BIST
                                ! didn't start. Should
                                ! probably log the
                                ! failure.

#else
        ba,a     internal_post
#endif
```



## 14.2 Power-On Self-Test (POST)

A POST tests parts of the processor and system that BIST did not test.

### 14.2.1 POSTs of Internal Storage Devices

BIST does not test the internal memory arrays in the SSP. The internal storage on which additional testing should be performed include:

- ☐ Windowed register file.
- ☐ Instruction cache data.
- ☐ Instruction cache tags.
- ☐ Data cache data.
- ☐ Data cache tags.
- ☐ Store Buffer.
- ☐ TLB.

The following subsections contain examples of code segments for testing the internal storage of the processor.

#### 14.2.1.1 Windowed Register File

Testing of the internal memory arrays of the processor starts with a rudimentary test of the windowed registers (Example 14–5). The routine tests the registers in a window, then moves to the next window. This test sets and checks several registers simultaneously to avoid activating bypass paths. The test detects all stuck-bit faults, most bit-line shorts, and many address-decoding faults. The %g registers are not explicitly tested.

#### Example 14–5. Windowed Register File Test

```
internal_post:
                                                    ! test machine
                                                    ! start by testing
                                                    ! windowed regs
                                                    ! run this with
                                                    ! ACTION.MIX=0 or may
                                                    ! test bypassing

test_reg_windows:
    set    0x1087,%g7                ! PSR with CWP=7
    set    0x1080,%g2                ! loop limit
```

```

        set      -1,%g6
        set      0x55555555,%g5
        set      0xAAAAAAAA,%g4
test_window_loop:
        mov      %g7,%psr                ! set the CWP
                                           ! good value for WIM is
                                           ! 0, no traps
        mov      %g0,%wim                ! set the WIM
        mov      %g6,%o0                ! initialize to F's
        mov      %g6,%o1                ! initialize to F's
        mov      %g6,%o2                ! initialize to F's
        mov      %g6,%o3                ! initialize to F's
        mov      %g6,%o4                ! initialize to F's
        mov      %g6,%o5                ! initialize to F's
        mov      %g6,%o6                ! initialize to F's
        mov      %g6,%o7                ! initialize to F's
        mov      %g6,%l0                ! initialize to F's
        mov      %g6,%l1                ! initialize to F's
        mov      %g6,%l2                ! initialize to F's
        mov      %g6,%l3                ! initialize to F's
        mov      %g6,%l4                ! initialize to F's
        mov      %g6,%l5                ! initialize to F's
        mov      %g6,%l6                ! initialize to F's
        mov      %g6,%l7                ! initialize to F's
test_o0_o1_o2_o3:
        cmp      %o0,%g6                ! check %o0
        bne      tests_bad              ! != so test is bad
        mov      %g5,%o0                ! 5's to %o0
        cmp      %o1,%g6                ! check %o1
        bne      tests_bad              ! != so test is bad
        mov      %g5,%o1                ! 5's to %o1
        cmp      %o2,%g6                ! check %o2
        bne      tests_bad              ! != so test is bad
        mov      %g5,%o2                ! 5's to %o2
        cmp      %o3,%g6                ! check %o3
        bne      tests_bad              ! != so test is bad
        mov      %g5,%o3                ! 5's to %o3
        cmp      %o0,%g5                ! check %o0
        bne      tests_bad              ! != so test is bad
        mov      %g4,%o0                ! A's to %o0
        cmp      %o1,%g5                ! check %o1
        bne      tests_bad              ! != so test is bad
        mov      %g4,%o1                ! A's to %o1
        cmp      %o2,%g5                ! check %o2
        bne      tests_bad              ! != so test is bad

```

## Power-On Self-Test (POST)

```
mov    %g4,%o2                ! A's to %o2
cmp    %o3,%g5                ! check %o3
bne    tests_bad              ! != so test is bad
mov    %g4,%o3                ! A's to %o3
cmp    %o0,%g4                ! check %o0
bne    tests_bad              ! != so test is bad
mov    %g0,%o0                ! zero %o0
cmp    %o1,%g4                ! check %o1
bne    tests_bad              ! != so test is bad
mov    %g0,%o1                ! zero %o1
cmp    %o2,%g4                ! check %o2
bne    tests_bad              ! != so test is bad
mov    %g0,%o2                ! zero %o2
cmp    %o3,%g4                ! check %o3
bne    tests_bad              ! != so test is bad
mov    %g0,%o3                ! zero %o3
test_o4_o5_o6_o7:
cmp    %o4,%g6                ! check %o4
bne    tests_bad              ! != so test is bad
mov    %g5,%o4                ! 5's to %o4
.
.
.
Continue in the same way, testing %o4, %o5, %o6, and %o7, then %l0, %l1,
%l2, and %l3, followed by %l4, %l5, %l6, and %l7.
.
.
.
cmp    %l7,%g4                ! check %l7
bne    tests_bad              ! != so test is bad
mov    %g0,%l7                ! zero %l7
test_loop_tail:
.
.
.
or     %o0,%o1,%o0            ! is any %o? non-zero
or     %o0,%o2,%o0
or     %o0,%o3,%o0
or     %o0,%o4,%o0
or     %o0,%o5,%o0
or     %o0,%o6,%o0
orcc   %o0,%o7,%o0
bne    tests_bad              ! not 0 so tests bad
nop                                ! is any %l? non-zero
```

```

or      %l0,%l1,%o0
or      %o0,%l2,%o0
or      %o0,%l3,%o0
or      %o0,%l4,%o0
or      %o0,%l5,%o0
or      %o0,%l6,%o0
orcc    %o0,%l7,%o0
bne     tests_bad                ! not 0 so tests bad
nop
add     %g7,-1,%g7                ! update CWP
cmp     %g7,%g2                  ! test for window 0
bge     test_window_loop
nop
ba,a    i_cache_test

```

#### 14.2.1.2 Instruction Cache Data

Example 14-6 contains a routine for testing the instruction cache data array. This test uses two helper routines, described in Example 14-9 and Example 14-10.

- 1) The first step is to write 0xAA in every byte of the cache data storage and verify that it is still 0xAA when read.
- 2) Next, step 1 is repeated with 0x55 written into and read from each byte.
- 3) The next step is to write a value based on its address in each cache doubleword and verify that it can be read back. This tests address decoding.
- 4) Step 3 is performed with the address-dependent data inverted.
- 5) The final step is to perform step 1 with 0x00 written into and read from each byte. This leaves the cache containing only zeros.

### Example 14–6. Instruction Cache Data Test

```
i_cache_test:
    set     0xAAAAAAAA, %o0
    call    i_cache_fill_test
    mov     %o0, %o1
    set     0x55555555, %o0
    call    i_cache_fill_test
    mov     %o0, %o1
    call    i_cache_unique_test
    nop
    set     0x0, %o0
    call    i_cache_fill_test
    mov     %o0, %o1
    ba, a   d_cache_test
```

#### 14.2.1.3 Instruction Cache Tags

This is left as an exercise to the user.

#### 14.2.1.4 Data Cache Data

The format for the data cache data test (see Example 14–7) is the same as for the instruction cache data. It also uses two helper routines, described in Example 14–11 and Example 14–12. The data cache is also left containing only zeros.

### Example 14–7. Data Cache Data Test

```
d_cache_test:
    set     0xAAAAAAAA, %o0
    call    d_cache_fill_test
    mov     %o0, %o1
    set     0x55555555, %o0
    call    d_cache_fill_test
    mov     %o0, %o1
    call    d_cache_unique_test
    nop
    set     0x0, %o0
    call    d_cache_fill_test
    mov     %o0, %o1
    ba, a   sb_test
```

#### 14.2.1.5 Data Cache Tags

This is left as an exercise to the user.

**14.2.1.6 Store Buffer Test**

This is left as an exercise to the user.

**14.2.1.7 TLB Test**

This is left as an exercise to the user.

**14.2.1.8 Other POSTs**

This concludes the POSTs of the internal storage devices.

Next the external storage devices (including external cache and main memory) could be tested. See Example 14–8.

When done with POST, branch to `reset_states` to set a standard operating state.

**Example 14–8. Other POSTs**

```

sb_test:
! none here

tlb_test:
! none here

internal_post_done:
external_post:
! none here yet

external_post_done:
! then complete reset
! for standard
! operation

        ba,a    reset_states

tests_bad:
! If reach here, some
! register has tested
! bad. Need to try to
! do some sort of
! system logging.
```

**14.2.2 POST Support Routines**

The following routines are called from the POST codes in Subsection 14.2.1. They consolidate portions of the tests that are used more than once.

These are leaf routines and therefore do not use `SAVE` or `RESTORE` instructions. They accept input operands in the `%o` registers and use only `%l` and `%o` registers for temporary storage.

### 14.2.2.1 Instruction Cache Support Routines

The routine in Example 14–9 is the instruction cache data fill test. Its argument is a doubleword in %o0 and %o1, which is a pattern. This pattern is stored into each doubleword of the instruction cache. After all doublewords are filled, each doubleword is read to check the integrity of the pattern. If any mismatch is found, the test branches to tests\_bad.

#### Example 14–9. Instruction Cache Fill Test

```

i_cache_fill_test:

                                ! %o0, %o1 is fill
                                ! pattern
                                ! use only %l? and %o?
                                ! registers
                                ! BLOCK_BASE_ADDR
                                ! BLOCK_STEP
                                ! BLOCK_LIMIT (5)
                                ! DWORD_BASE_ADDR
                                ! DWORD_STEP
                                ! DWORD_LIMIT

                                set    0x0,%l0
                                set    0x04000000,%l1
                                set    0x14000000,%l2
                                set    0x0,%l3
                                set    0x8,%l4
                                set    0x00001000,%l5

i_c_line_st_loop:
i_c_word_st_loop:

                                ! store pattern in data
                                ! dword

                                stda   %o0,[%l0+%l3] 0x0d
                                add    %l3,%l4,%l3
                                cmp    %l3,%l5
                                bl     i_c_word_st_loop
                                nop

                                ! move to next line

                                add    %l0,%l1,%l0
                                cmp    %l0,%l2
                                bl     i_c_line_st_loop
                                mov    %g0,%l3

                                set    0x0,%l0

                                ! reset word counter
                                ! stores are done, now
                                ! compare
                                ! reset line counter

i_c_line_cmp_loop:
i_c_word_cmp_loop:

                                ! load contents of
                                ! dword into %l6 %l7

```

```

        ldda    [%10+%13] 0x0d,%16
        cmp     %o0,%16
        bne     tests_bad
        nop
        cmp     %o1,%17
        bne     tests_bad
        nop
                                ! update word counter
        add     %13,%14,%13
        cmp     %13,%15
        bl      i_c_word_cmp_loop
        nop
                                ! update line counter
        add     %10,%11,%10
        cmp     %10,%12
        bl      i_c_line_cmp_loop
        mov     %g0,%13                                ! reset word counter
        retl
        nop

```

The routine in Example 14-10 is a test for instruction data array cache address uniqueness. This routine has no arguments. It stores every doubleword of the instruction cache with a doubleword composed of the word's address in the low word; it stores the complement of its address in the upper doubleword. When every cache location has been written, each is read and checked for corruption. Any mismatch causes a transfer to `bad_test`. The test runs twice; all data is complemented on the first iteration and uncomplemented on the second.

#### Example 14-10. Instruction Cache Address Uniqueness Test

```

i_cache_unique_test:
                                ! use only %l? and %o?
                                ! registers
                                ! LINE_BASE_ADDR
                                ! LINE_STEP
                                ! LINE_LIMIT (5)
                                ! DWORD_BASE_ADDR
                                ! DWORD_STEP
                                ! DWORD_LIMIT
                                ! COMPLEMENT

        set     0x0,%10
        set     0x04000000,%11
        set     0x14000000,%12
        set     0x0,%13
        set     0x8,%14
        set     0x00001000,%15
        set     -1,%o3

i_c_u_complement_loop:
i_c_u_line_st_loop:

```



## Power-On Self-Test (POST)

---

```
i_c_u_word_st_loop:
    add    %l0,%l3,%o0
    orn    %g0,%o0,%o1
    xor     %o0,%o3,%o0
    xor     %o1,%o3,%o1
                                           ! store pattern in data
                                           ! dword

    stda    %o0,[%l0+%l3] 0x0d
    add     %l3,%l4,%l3
    cmp     %l3,%l5
    bl      i_c_u_word_st_loop
    nop

                                           ! move to next line

    add     %l0,%l1,%l0
    cmp     %l0,%l2
    bl      i_c_u_line_st_loop
    mov     %g0,%l3
                                           ! reset word counter
                                           ! stores are done, now
                                           ! compare
                                           ! reset line counter

    set     0x0,%l0

i_c_u_line_cmp_loop:
i_c_u_word_cmp_loop:
    add     %l0,%l3,%o0
    orn     %g0,%o0,%o1
    xor     %o0,%o3,%o0
    xor     %o1,%o3,%o1
                                           ! load contents of
                                           ! dword into %l6 %l7

    ldda    [%l0+%l3] 0x0d,%l6
    cmp     %o0,%l6
    bne     tests_bad
    nop
    cmp     %o1,%l7
    bne     tests_bad
    nop

                                           ! update word counter

    add     %l3,%l4,%l3
    cmp     %l3,%l5
    bl      i_c_u_word_cmp_loop
    nop

                                           ! update line counter

    add     %l0,%l1,%l0
    cmp     %l0,%l2
    bl      i_c_u_line_cmp_loop
```

```

mov    %g0,%l3                ! reset word counter

    cmp    %o3,%g0
    bne    i_c_u_complement_loop
    mov    %g0,%o3

    retl
    nop

```

#### 14.2.2.2 Data Cache Support Routines

The routine in Example 14-11 is the data cache data array fill test. Its argument is a doubleword in %o0 and %o1, which is a pattern. This pattern is stored in each doubleword of the data cache. After all doublewords are filled, each doubleword is read to check the integrity of the pattern. If any mismatch is found, the test branches to tests\_bad.

#### Example 14-11. Data Cache Fill Test

```

d_cache_fill_test:

                                ! %o0, %o1 is fill
                                ! pattern
                                ! use only %l? and %o?
                                ! registers
                                ! LINE_BASE_ADDR
                                ! LINE_STEP
                                ! LINE_LIMIT (5)
                                ! DWORD_BASE_ADDR
                                ! DWORD_STEP
                                ! DWORD_LIMIT

    set    0x0,%l0
    set    0x04000000,%l1
    set    0x10000000,%l2
    set    0x0,%l3
    set    0x8,%l4
    set    0x00001000,%l5

d_c_line_st_loop:
d_c_word_st_loop:

                                ! store pattern in data
                                ! dword

    stda    %o0,[%l0+%l3] 0x0f
    add     %l3,%l4,%l3
    cmp     %l3,%l5
    bl     d_c_word_st_loop
    nop

                                ! move to next line

    add     %l0,%l1,%l0
    cmp     %l0,%l2
    bl     d_c_line_st_loop
    mov     %g0,%l3

                                ! reset word counter
                                ! stores are done, now

```

## Power-On Self-Test (POST)

---

```
                                ! compare
                                ! reset line
        set      0x0,%10
counterd_c_line_cmp_loop:
d_c_word_cmp_loop:

                                ! load contents of
                                ! dword into %16 %17
        ldda     [%10+%13] 0x0f,%16
        cmp      %o0,%16
        bne      tests_bad
        nop
        cmp      %o1,%17
        bne      tests_bad
        nop

                                ! update word counter
        add      %13,%14,%13
        cmp      %13,%15
        bl       d_c_word_cmp_loop
        nop

                                ! update line counter
        add      %10,%11,%10
        cmp      %10,%12
        bl       d_c_line_cmp_loop
        mov      %g0,%13
                                ! reset word counter
        retl
        nop
```

The routine in Example 14-12 is a test for data cache data array address uniqueness. This routine has no arguments. It stores every doubleword of the data cache with a doubleword composed of the word's address in the low word; it stores the complement of its address in the upper doubleword. When every cache location has been written, each is read and checked for corruption. Any mismatch causes a branch to `tests_bad`. The test runs twice; all data is complemented on the first iteration and uncomplemented on the second.

### Example 14-12. Data Cache Address Uniqueness Test

```
d_cache_unique_test:

                                ! use only %1? and %o?
                                ! registers
                                ! LINE_BASE_ADDR
        set      0x0,%10
                                ! LINE_STEP
        set      0x04000000,%11
                                ! LINE_LIMIT (5)
        set      0x10000000,%12
                                ! DWORD_BASE_ADDR
        set      0x0,%13
                                ! DWORD_STEP
        set      0x8,%14
```

```

        set      0x00001000,%15          ! DWORD_LIMIT
        set      -1,%o3                  ! COMPLEMENT

d_c_u_complement_loop:
d_c_u_line_st_loop:
d_c_u_word_st_loop:
        add      %l0,%l3,%o0
        orn      %g0,%o0,%o1
        xor      %o0,%o3,%o0
        xor      %o1,%o3,%o1

                                           ! store pattern in data
                                           ! dword

        stda     %o0,[%l0+%l3] 0x0f
        add      %l3,%l4,%l3
        cmp      %l3,%l5
        bl       d_c_u_word_st_loop
        nop

                                           ! move to next line

        add      %l0,%l1,%l0
        cmp      %l0,%l2
        bl       d_c_u_line_st_loop
        mov      %g0,%l3

                                           ! reset word counter
                                           ! stores are done, now
                                           ! compare
        set      0x0,%l0                  ! reset line counter

d_c_u_line_cmp_loop:
d_c_u_word_cmp_loop:
        add      %l0,%l3,%o0
        orn      %g0,%o0,%o1
        xor      %o0,%o3,%o0
        xor      %o1,%o3,%o1

                                           ! load contents of
                                           ! dword into %l6 %l7

        ldda     [%l0+%l3] 0x0f,%l6
        cmp      %o0,%l6
        bne      tests_bad
        nop
        cmp      %o1,%l7
        bne      tests_bad
        nop

                                           ! update word counter

        add      %l3,%l4,%l3
        cmp      %l3,%l5
        bl       d_c_u_word_cmp_loop

```

## Power-On Self-Test (POST)

---

```

nop
                                     ! update line counter
    add    %10,%11,%10
    cmp    %10,%12
    bl     d_c_u_line_cmp_loop
    mov    %g0,%13                   ! reset word counter

    cmp    %o3,%g0
    bne    d_c_u_complement_loop
    mov    %g0,%o3
    retl
nop
```

## 14.3 Reset States

This routine resets the processor's internal states for normal operation.

### 14.3.1 PSR

First set the PSR (see Section 4.2) to some standard state as defined for a particular system.

The following are guidelines for setting the PSR on startup:

- 1) Set S to enable the reset handler to perform LDA and STA instructions. PS may be clear to facilitate transferring into user mode with a RETT instruction at the end of the reset handler.
- 2) ET should be clear because the state isn't set up yet for interrupts or traps.
- 3) Set CWP to a convenient value.
- 4) EC is clear because SuperSPARC doesn't have a coprocessor.
- 5) EF is set so that the Floating Point Unit (FPU) can be initialized.
- 6) ICC is set to any convenient value.
- 7) Set the WIM and TBR registers to values that work in this system.
- 8) Enable superscalar execution (Example 14-13), which allows the rest of the reset handler to run faster.

### 14.3.2 Reset States: Control Registers and Superscalar Execution

Example 14-13 shows an example of the portion of a reset handler that sets the initial values in the important system control registers. In the SSP, this includes enabling the execution of more than one instruction per cycle (superscalar execution).

The registers that should be initialized are:

- ☐ Processor State Register (PSR).
- ☐ Trap Base Register (TBR).
- ☐ Window Invalid Mask (WIM).

Superscalar execution is enabled by setting the multiple instruction execution (MIX) bit in the breakpoint action register (ACTION).

#### *Example 14-13. Reset States: Control Registers and Superscalar Execution*

```
reset_states:
#define INIT_PSR 0x00001087          ! CWP=7, S=1, EF=1
#define INIT_WIN 0x00000001
```

```
init_cregs:
    set     INIT_PSR, %g1
    mov     %g1, %psr                ! Initialize PSR
    set     INIT_TBR, %g1
    mov     %g1, %tbr                ! Initialize TBR
    set     INIT_WIM, %g1
    mov     %g1, %wim                ! Initialize WIM
enable_superscalar_exec:
    set     0x0, %g1                 ! ECNTL_VADDR
    set     0x01000, %g2             ! ECNTL_DATA
    sta     %g2, [%g1] 0x4C          ! ECNTL_ASI

    ba, a    clear_mmu_and_caches
```

### 14.3.3 Initialization of MMU and Caches

The fault status register (MFSR), translation lookaside buffer (TLB), and caches must be cleared before the MMU or caches can be enabled. This procedure is as follows:

- 1) Clear the MFSR by reading it.
- 2) Invalidate the TLB entries with a demap\_all operation.
- 3) Clear the instruction cache valid bits and lock bits with instruction cache flash clear.
- 4) Clear the data cache valid and lock bits with data cache flash clear.
- 5) Initialize the MMU context register and MMU context table pointer to values suitable for use in the system.

CTP\_VALUE is the physical address of the context table in memory (see Subsection 9.12.2).

Example 14-14 shows an example of code to initialize the MMU and caches.

#### Example 14-14. MMU and Cache Initialization

```
clear_mmu_and_caches:
clear_mmu_mfsr_reg:
    set     0x0300, %g1                ! MFSR_VADDR
                                           ! Clear the MFSR by
                                           ! reading it
    lda     [%g1] 0x04, %g0            ! MFSR_ASI
demap_all_tlb_entries:
```

```

        set      0x0400,%g1                ! MENTIRE_FLUSH_VADDR
                                           ! Demap all entries
        sta      %g0,[%g1] 0x03            ! MENTIRE_FLUSH_ASI
clear_icache_valid_bits:
        set      0x0,%g1                  ! CENTIRE_FLUSH_VADDR
                                           ! Clear all valid-bits
        sta      %g0,[%g1] 0x36            ! CENTIRE_FLUSH_ASI
clear_icache_lock_bits:
        set      0x8000000,%g1            ! CENTIRE_LOCK_FLUSH_VADDR
                                           ! Clear all LOCK-bits
        sta      %g0,[%g1] 0x36            ! CENTIRE_LOCK_FLUSH_ASI
clear_dcache_valid_bits:
        set      0x0,%g1                  ! DENTIRE_FLUSH_VADDR
                                           ! Clear all valid-bits
        sta      %g0,[%g1] 0x37            ! DENTIRE_FLUSH_ASI
clear_dcache_lock_bits:
        set      0x80000000,%g1           ! DENTIRE_LOCK_FLUSH_VADDR
                                           ! Clear all LOCK-bits
        sta      %g0,[%g1] 0x37            ! DENTIRE_LOCK_FLUSH_ASI
init_mmu_context_reg:
        set      0x0200,%g1                ! MCONTEXT_VADDR
        set      USER_CONTEXT_NUMBER,%g2
        sta      %g2,[%g1] 0x04            ! MCONTEXT_ASI
init_mmu_ctp_reg:
        set      0x0100,%g1                ! MCTP_VADDR
        set      CTP_VALUE,%g2
        sta      %g2,[%g1] 0x04            ! MCTP_ASI
        ba, a    init_mcc_control_reg

```

#### 14.3.4 MultiCache Controller

If the configuration contains a MultiCache Controller, it must be initialized. The code in Example 14–15 initializes the MXCC Control register and MXCC Interrupt Mask register.

The value of the MXCC\_CNTL\_DATA\_WORD is system-dependent. Following are some suggested starting-point values for an MBus system.

- ☐ RC = 0      count both read and write references
- ☐ BWC = 0      no BW's connected\*
- ☐ WI = 0      no write invalidate\*
- ☐ PF = 1      prefetching enabled
- ☐ MC = 1      multiple commands enabled



- ☐ PE = 1 parity enabled
- ☐ CE = 1 E-cache enabled
- ☐ CS = 0 1 MB of cache\*
- ☐ HC = 0 not half-sized cache\*

\* Ignored in MBus configuration.

Following are some suggested starting-point values for an XBus system:

- ☐ RC = 0 count both read and write references
- ☐ BWC BW count set to the number of BWs connected
- ☐ WI = 0 write invalidate enabled.
- ☐ PF = 1 prefetching enabled
- ☐ MC = 1 multiple command execution enabled.
- ☐ PE = 1 parity enabled
- ☐ CE = 1 external cache enabled
- ☐ CS = 0 1 MB E-cache
- ☐ HC = 0 not half-sized E-cache

## Example 14–15. MXCC Initialization

```
#ifdef MXCC

#define MXCC_CNTL_DATA_WORD_MBUS 0x0000003c

#define MXCC_CNTL_DATA_WORD_XBUS 0x0000017c      ! 2 BW's
#define MXCC_CNTL_DATA_WORD      MXCC_CNTL_DATA_WORD_MBUS
init_mcc_control_reg:
    set      MXCC_CNTL_DATA_WORD, %g1
    set      0x01c00200,%g2                      ! MXCC_CNTL_VADDR
    stda     %g0,[%g2] 0x02                      ! MXCC_CNTL_ASI
init_mcc_int_mask:
                                                ! allow all MXCC XBus
                                                ! interrupts
                                                ! they won't be
                                                ! recognized until
                                                ! PSR.ET=1
    set      0x01c00500,%g2                      ! MXCC_INT_MASK_ADDR
```

```

mov      0x0, %g1                ! zero in %g0,%g1 pair
stda     %g0, [%g2] 0x02         ! MXCC_CNTL_ASI
#endif

```

### 14.3.5 MMU Control Register

Next the MMU control register (see Example 14–16) must be initialized. The value for MCNTL depends on the MMU function desired in the system (see Subsection 9.12.1).

The following subsections contain some suggested starting-point values for the MMU control register.

#### MCNTL.TC and MCNTL.AC

- ☐ TC = 1      MMU tablewalks cacheable in the external cache
- ☐ AC = 0      alternate accesses not cacheable

MCNTL.TC and MCNTL.AC values depend on how software accesses the in-memory page tables and other data in alternate spaces.

The table walk cacheable bit indicates external cacheability, even though table walk data is never cached internally.

The AC bit should be set to the same value as the C bit field of the store buffer tag register (since this is the state of the C bit in the original transaction).

#### MCNTL.BT

- ☐ BT = 0      boot mode disabled

Reset has initialized MCNTL.BT to zero, forcing boot mode translations in the MMU for instruction fetch (see Section 9.10). This point in the reset handler may be too early for some systems to disable boot mode. In many systems, the contents of the boot PROM may need to be copied somewhere else and some MMU mappings established before switching off boot mode.

#### MCNTL.PE and MXCCCR.PE

- ☐ PE = 1      parity generation/checking enabled

MCNTL.PE should match MXCCCR.PE.

#### MCNTL.PSO

- ☐ PSO = 0      TSO

Whether PSO is recommended depends on whether the software can accommodate it. See Chapter 8.

**MCNTL.SE, MCNTL.SB, MCNTL.IE, and MCNTL.DE**

- ☐ SE = 1      snooping enabled
- ☐ SB = 1      store buffer enabled
- ☐ IE = 1      I-cache enabled
- ☐ DE = 1      D-cache enabled

It is generally recommended that snooping, the store buffer, and the caches be enabled. See Chapter 10.

**MCNTL.NF and MCNTL.EN**

- ☐ NF = 0      no-fault mode disabled
- ☐ EN = 1      MMU enabled

**Note:**

In order for the MMU to be enabled, the page tables, context tables, context table pointer, and context register must be set up. See Section 9.2.

**IFLUSH Instruction**

After storing the MCNTL register with changes affecting instruction access, issue an IFLUSH instruction to synchronize instruction fetch with the change to MCNTL. A JMP instruction may be included between the STA to MCNTL and the IFLUSH to transfer to a new address synchronously with the change in MMU modes.

**Example 14–16. MMU Control Register**

```

! NOTE BT=1

#define MCNTL_DATA_WORD 0x00017f01

init_mmu_mcntl_reg:
    set    0x0,%g1                ! MCNTL_VADDR
    set    MCNTL_DATA_WORD,%g2
                                ! init MCNTL
    sta    %g2,[%g1] 0x04         ! MCNTL_ASI
    iflush %g0                    ! <<<***** IMPORTANT ***
/* New MMU modes take effect with the next instruction
   after the IFLUSH. MCNTL.BT could be off here so the
   next instruction could be from a non-contiguous
   address. Use caution!
*/

```

### 14.3.6 Floating-Point Unit

Next the floating-point unit is initialized. The new FSR value can be loaded only from memory. (See Example 14–17.)

#### Example 14–17. Floating-Point Unit Initialization

```
#define INIT_FSR 0x0F800000                                ! TEM=0x1f
                                                         ! address of data word
                                                         ! can read and write

#define FSR_SCRATCH_DATA_ADDR 0x000F0000

init_fpu_fsr:
    set    FSR_SCRATCH_DATA_ADDR, %g2
    set    INIT_FSR, %g1
    st     %g1, [%g2]                                     ! store init data
    ld     [%g2], %fsr                                    ! Initialize FSR
```

### 14.3.7 End Reset

Example 14–18 shows a JMP/RETT pair that transfers to ENTRY\_POINT with S set to PS and ET set.

#### Example 14–18. End Reset

```
end_reset:

                                                         ! Go do some work.
                                                         ! Below example uses
                                                         ! JMP/RETT
                                                         ! to go to user mode
                                                         ! program. S <- PS and
                                                         ! ET <- 1
                                                         ! ENTRY_POINT points to
                                                         ! program entry point

    set    ENTRY_POINT, %12
    jmp    %12                                           ! Start the user
                                                         ! program

    rett   %12 + 4
```



## **Diagnostic Operation**

---

---

The SuperSPARC processor (SSP) implements software-debugging capabilities and external monitors. These enable the user to examine the SSP at each cycle and aid in system debug.

<b>Topic</b>	<b>Page</b>
<b>15.1 Software Debugging Facilities</b> .....	<b>15-2</b>
<b>15.2 Counters and Breakpoints</b> .....	<b>15-6</b>
<b>15.3 External Monitors</b> .....	<b>15-13</b>

## 15.1 Software Debugging Facilities

The SSP provides debugging capabilities to facilitate debugging software and hardware prototypes. Four types of breakpoint mechanisms are provided:

- ☐ Code address,
- ☐ Data address,
- ☐ Instruction count underflow, and
- ☐ Cycle count underflow.

These mechanisms can be selectively enabled to generate either `data_access_exceptions` or `instruction_access_exceptions`. They can be programmed to generate a selectable interrupt. In addition, they can be set to activate an external pin to easily trigger external analysis equipment. They are also a fundamental part of SuperSPARC's scan-based debug features, described in Chapter 22.

In addition to address breakpoint facilities, programmable timers are provided for debug, code profiling, and performance analysis. They can provide a high-precision, low-overhead timer for small-application benchmarking. Their major application is to assist development or diagnostic teams in early hardware and software debug.

### 15.1.1 Address Breakpoints—Code (Instruction) or Data

A single code (instruction) or data breakpoint register is available. This breakpoint can match on either 32-bit virtual or 36-bit physical addresses for code or data. When a breakpoint is set on an instruction, the instruction will not be executed. This holds true for fault, interrupt, and scan-based debug.

---

**Note:**

A maximum of one code space breakpoint or one data space breakpoint can be active at a given time.

---

Each bit in the bitwise address comparison can be masked off separately to force equality on that bitwise comparison. The address equality bitmasks can be used to find references within a particular segment, page, cache line, or word independent of the size of the access. Data space breakpoints can be qualified with access type (reads-only, writes-only, read-or-write). Atomic references (e.g., SWAP/LDSTUB) are considered both read and writes.

The action-upon-event control register (ACTION) specifies whether the address breakpoint should generate an exception or interrupt or activate the external strobe (ESB) pin. The action-on-event register is defined in Subsection 15.2.5.

### 15.1.2 Counter Breakpoints - Code (Instruction) or Cycle

A single 32-bit-wide control register programs the two 16-bit counters for instruction and cycle counts at the same time. It is not possible to modify one counter without the other.

---

**Note:**

A maximum of one instruction count breakpoint and one cycle count breakpoint can be active at a given time.

---

The 16-bit cycle counter will count up to about 1.3 milliseconds at 50 MHz. Longer-duration counters must be simulated in software by accumulating underflows of the 16-bit counter into a larger counter in memory.

The instruction counter can count either faster or slower than the cycle counter, depending on the execution characteristics of the processor. It could theoretically count three times faster, if SuperSPARC were continuously executing three instruction groups. In general, it will count slightly faster than the cycle counter.

The combination of these two counter interrupts can be used to calculate the dynamic performance (in million instructions per cycle (MIPS) of the executing program.

Cycle counter underflow events are always reported to the last valid instruction in the current instruction group. Instruction counter events occur as interrupts or scan-based debug requests to the instruction after the instruction that causes the counter event. Instruction counter expiration events will be reported after the first instruction that causes the underflow.

Once set, the counter expiration event is persistent until served. The instruction that caused the counter event will ultimately be restarted.

When a cycle counter expires, action for that event may be deferred until valid instructions are available and the pipeline is able to progress. If the action is deferred, the request (interrupt or scan-based debug) will persist until the pipeline is able to continue. Once the initial event is signalled, the cycle counter continues counting down through the most positive number. In this way the total number of elapsed cycles from a given point may be calculated.

The instruction counter decrements the existing instruction count by the number of instructions that complete execution in a given cycle. The number can vary anywhere from 0 to 3 instructions per cycle. The cycle counter always decrements by 1.

### 15.1.3 Setting Up Breakpoints

Table 15-1 describes how to set up breakpoints using the proper control registers and where to find information that the actual action(s) has/have been triggered after the breakpoint.



Table 15-1. Breakpoints—Control and Status

			CONTROL										STATUS																		
			BKC						ACTION						CTRC	CTRS	BKS		MSTAT												
			CSPACE	PAMD	CBFEN	CBKEN	DBFEN	DBREN	DBWEN	MIX	BCIPL	STEN_CBK	STEN_ZIC	STEN_DBK	STEN_ZCC	IEN_CBK	IEN_ZIC	IEN_DBK	IEN_ZCC	ICNTEN	CCNTEN	ZICIS	ZCCIS	CBKIS	CBKFS	DBKIS	DBKFS	CBKM	DBKM	ZICM	ZCCM
Address Breakpoints	Data	RD	data_acc_exc	0	x	0	0	1	1	0	x	x	x	x	x	x	x	x	x	x	x	x	u	u	u	u	1	u	u	u	u
		WR	data_acc_exc	0	x	0	0	1	0	1	x	x	x	x	x	x	x	x	x	x	x	x	u	u	u	u	1	u	u	u	u
		RD	Interrupt	0	x	0	0	0	1	0	x	S	x	x	x	x	x	1	x	x	x	u	u	u	1	u	u	u	u	u	u
		WR	Interrupt	0	x	0	0	0	0	1	x	x	x	x	x	x	x	1	x	x	x	u	u	u	1	u	u	u	u	u	u
		RD	Emulation req	0	x	0	0	0	1	0	x	x	x	x	x	x	x	0	x	x	x	u	u	u	u	u	u	1	u	u	u
		WR	Emulation req	0	x	0	0	0	0	1	x	x	x	x	x	x	x	0	x	x	x	u	u	u	u	u	u	1	u	u	u
		RD	ESTROBE	0	x	0	0	x	1	1	x	x	x	x	1	x	x	x	x	x	x	u	u	u	E	E	u	E	u	u	u
	Code		instr_acc_exc	1	x	1	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	u	u	u	1	u	u	u	u	u	
			interrupt	1	x	0	1	x	x	0	x	S	x	x	x	x	1	x	x	x	x	u	u	1	u	u	u	u	u	u	u
			emulation req	1	x	0	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	u	u	u	u	u	u	1	u	u	u
			ESTROBE	1	x	x	1	x	x	x	x	x	1	x	x	x	x	x	x	x	x	u	u	E	E	u	u	E	u	u	u
Counter Breakpoint	Data		interrupt	x	x	x	x	x	x	x	x	S	x	x	x	x	x	x	1	x	1	u	1	1	u	u	u	u	u	u	
			emulation req	x	x	x	x	x	x	x	x	x	x	x	x	x	x	0	x	1	1	u	u	u	u	u	u	u	u	1	u
			ESTROBE	x	x	x	x	x	x	x	x	x	x	x	1	x	x	x	x	1	1	u	u	E	u	u	u	u	u	E	u
			ESTROBE	x	x	x	x	x	x	x	x	x	x	1	x	x	x	x	x	1	1	u	u	u	u	u	u	u	u	1	u
	Code		interrupt	x	x	x	x	x	x	x	x	S	x	x	x	x	x	1	x	x	1	1	u	u	u	u	u	u	u	u	u
			emulation req	x	x	x	x	x	x	x	x	x	x	x	x	x	0	x	x	1	1	u	u	u	u	u	u	u	1	u	u
			ESTROBE	x	x	x	x	x	x	x	x	x	1	x	x	x	x	x	1	1	u	E	u	u	u	u	u	u	E	u	u

† S = Set BCIPL to select IRL; x = don't care; E = Event Dependent; u = unchanged

In Table 15-1, the notations S, x, and u are self-explanatory. The notation E (which is given to an affected status bit) indicates that the bit is Event-dependent, meaning that the bit status after a breakpoint depends on whether a particular event has occurred. For example, the BKS.DBKIS bit in row data address breakpoint ESB, which is designated the notation E, will be set if ACTION.IEN\_DBK and ACTION.BCIPL were set when the breakpoint occurred. (The latter two bits are designated "don't cares").

For all the scan-based debug request cases that are initiated by breakpoints, the MCMD.INITM bit must be set. See Chapter 22.

When setting a breakpoint, the programmer must allow for the latency until the breakpoint can be guaranteed active. To achieve this, the last STA 0x38 instruction that sets the breakpoint registers must be followed by an IFLUSH or a BA instruction.

Also, after the breakpoint has been taken, the programmer must explicitly disable (by disabling the appropriate control bits) that particular breakpoint, so that, when normal execution is resumed, the same breakpoint will not be taken again.

#### 15.1.4 Priorities of Debug Interrupt and Exception

All address breakpoint interrupts and counter underflow interrupts use a common user-programmable interrupt priority level (ACTION.BC IPL—see Subsection 15.2.5). Status bits are provided to differentiate between these interrupt sources. The following describes the priority order that these interrupts and exceptions adhere to.

- 1) If multiple breakpoint fault events (code, data) are signalled simultaneously, both code and data breakpoint status registers will set their fault status bits.
- 2) If multiple breakpoint interrupt events are signalled simultaneously, each activity will set its interrupt status bits.
- 3) Exception sources from multiple instructions are prioritized based on instruction order. An exception reported to an instruction will have higher priority than a simultaneous exception to the next instruction.
- 4) Asynchronous data\_access\_exceptions are honored before instruction\_access\_exceptions.
- 5) Instruction\_access\_exceptions are honored before synchronous data\_access\_exceptions, and all data\_access\_exceptions are honored before interrupts.

## 15.2 Counters and Breakpoints

### 15.2.1 MMU Breakpoint Control Registers ASI=0x38

There are four memory-mapped, SuperSPARC-specific, MMU-breakpoint diagnostic registers (see Table 15-2). These registers are double-word access only; any other size will cause a `data_access_exception`. All breakpoint enable and status bits are cleared at reset. All values and masks are unchanged through reset.

A single-address breakpoint is controlled by these four registers. The address breakpoint may be set in code space or data space, but not both simultaneously.

An Address Breakpoint can:

- ☐ Generate an instruction or data access breakpoint exception.
- ☐ Generate an instruction or data address breakpoint interrupt.
- ☐ Generate an instruction or data address breakpoint scan-based debug request.
- ☐ Enable the ESB pin.
- ☐ None of the above.

The response selected is determined by the breakpoint control register (BKC), the ACTION register, and the JTAG MCMD scan register. More details on the JTAG MCMD register can be found in Chapter 22.

Instruction and data address breakpoint shares the same breakpoint register set. The address map for these MMU diagnostic registers is shown in Table 15-2.

Table 15-2. MMU Diagnostic (Breakpoint) Registers

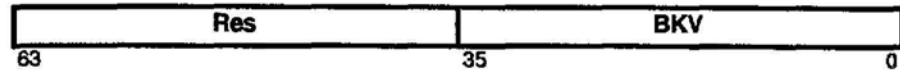
Addr<9:8>	Reg Name	Description	Reference
0	BKV	Breakpoint Value (Addr)	15.2.1.1
1	BKM	Breakpoint Mask	15.2.1.2
2	BKC	Breakpoint Control	15.2.1.3
3	BKS	Breakpoint Status	15.2.1.4

These registers are defined as follows:

#### 15.2.1.1 Breakpoint Value Register (BKV)

This register defines the code or data breakpoint address value.

Figure 15–1. Breakpoint Value Register (BKV)

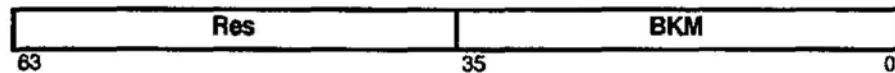


<b>res</b>	Reserved. These bits are ignored on write and read as zero.
<b>BKV</b>	Breakpoint Value. Contains the 36-bit value with which either physical or virtual address of the code or data being accessed will be compared (as determined by BKC.CSPACE in Subsection 15.2.1.3). When a match occurs, an event is generated depending on the state of the BKC and ACTION registers. These actions are also affected by the JTAG MCMD.INITM bits (not programmer-visible).

**15.2.1.2 Breakpoint Mask Register (BKM)**

This register defines a mask value useful for address-matching across a range.

Figure 15–2. Breakpoint Mask Register (BKM)

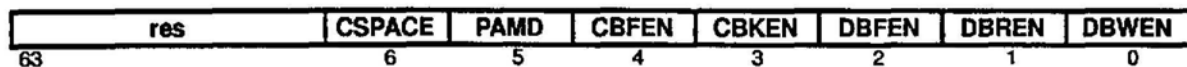


<b>res</b>	Reserved. These bits are ignored on write and read as zero.
<b>BKM</b>	Breakpoint Mask. This field defines a per-bit comparison mask for the BKV field. For any bit that is set in BKM, the equivalent bit in BKV is ignored in the address comparison. This can be used to match on ranges of addresses.

**15.2.1.3 Breakpoint Control Register (BKC)**

This register controls whether the breakpoint is to be set for code or data space access, and whether to compare a physical or virtual address. It also controls the enable for different types of breakpoints. All bits are cleared on both hardware and watchdog reset.

Figure 15–3. Breakpoint Control Register (BKC)



<b>res</b>	Reserved. This field is ignored on writes and read as zero.
<b>CSPACE</b>	Code Space Address. If CSPACE=1, the address in the BKV register is compared to code space address. DBREN, DBWEN and DBFEN are ignored. If CSPACE=0, the address in the BKV register is compared to data space address. CBKEN and CBFEN are ignored.
<b>PAMD</b>	Physical Address. If PAMD=1, the physical address is compared (BKV[35:0]). If PAMD=0, virtual address is compared (BKV[31:0]), ignore BKV[35:32]).
<b>CBFEN</b>	Enable Code Breakpoint Fault Generation. When this bit is set, a code breakpoint match will cause an instruction_access_exception. When this bit is cleared, an interrupt as defined in ACTION will be reported (see Subsection 15.2.5, Action Register).
<b>CBKEN</b>	Enable Code Breakpoints. If disabled, no code breakpoint can occur.
<b>DBFEN</b>	Enable Data Breakpoint Fault Generation. When this bit is set, a data breakpoint match will cause a data_access_exception. When this bit is cleared (and either DBREN or DBWEN is set), an interrupt as defined in ACTION register will be reported (see Subsection 15.2.5, Action Register).
<b>DBREN</b>	Enable Data Breakpoints for Read Transactions. This bit must be set (with CSPACE cleared) for data read breakpoints to occur.
<b>DBWEN</b>	Enable Data Breakpoints for Writes Transactions. This bit must be set (with CSPACE cleared) for data write breakpoints to occur.

#### 15.2.1.4 Breakpoint Status Register (BKS)

This register reports on the status of either code or data breakpoints. Any type of reset sequence or any load ASI from this register will clear all the bits in this register.

Table 15-3. Breakpoint Status Register (BKS)

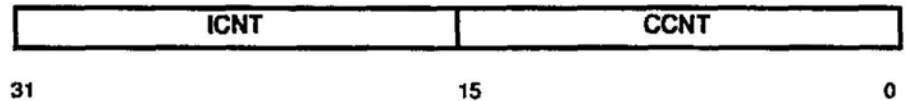
res	CBKIS	CBKFS	DBKIS	DBKFS
63	3	2	1	0

<b>CBKIS</b>	Code Breakpoint Interrupt. Indicates that an interrupt was generated as a result of a code breakpoint.
<b>CBKFS</b>	Code Breakpoint Access Exception. Indicates that a code access exception was generated as a result of a code breakpoint. This bit is also set on an instruction_access_exception.
<b>DBKIS</b>	Data Breakpoint Interrupt. Indicates that an interrupt was generated as a result of a data breakpoint.
<b>DBKFS</b>	Data Breakpoint Access Exception. Indicates that a data_access_exception was generated as a result of a data breakpoint. This bit is also set on a data_access_exception.

### 15.2.2 Counter Breakpoint Value Register (CTRV) ASI=0x49

The CTRV register is a register that holds the counter breakpoint values and is composed of the following two 16-bit fields:

Figure 15-4. Counter Breakpoint Value Register (CTRV)



**ICNT** Instructions Left. This field specifies the number of instructions left before the next counter event. (See Note below.)

**CCNT** Cycles Left. This field specifies the number of cycles left before the next counter event. (See Note below.)

Neither counter is decremented when SuperSPARC is in scan-based debug mode. ICNT and CCNT are read and written as a pair and are unchanged at reset.

**Note:**

The total number of instructions remaining to execute before the next counter event is given by the value held in CTRV.ICNT (readable by LDA 0x49) plus the number of instructions in the group prior to the group containing the LDA 0x49 instruction. If the group prior to the group containing the LDA 0x49 instruction has zero instructions, the value held by CTRV.ICNT reflects the exact number of instructions remaining to execute before the next counter event. An example is to use JMWL and BA branching to the LDA 0x49, which will force a bubble (hence zero instruction group) between the BA and the LDA. Another example might be using a sequential instruction before the LDA 0x49 or other methods that guarantee a group with a known number of instructions, allowing the programmer to get an exact instruction count until the counter event. Similarly, the number of cycles remaining before the next counter event is always one more than the value held in CTRV.CCNT (readable by LDA 0x49).

**15.2.3 Counter Breakpoint Control Register (CTRC) ASI=0x4a**

This register enables either the instruction counter or cycle counter. The number of instructions or cycles counted is specified in the CTRV register (see Subsection 15.2.2).

res	ICNTEN	CCNTEN
31	1	0

<b>res</b>	Reserved. This field is ignored on writes and read as zero.
<b>ICNTEN</b>	Enable Instruction Counter. This bit, when set, will enable the instruction counter in the CTRV register. This bit is cleared on either a power-on reset or watchdog reset.
<b>CCNTEN</b>	Enable Cycle Counter. This bit, when set, will enable the cycle counter in the CTRV register. This bit is cleared on either a power-on reset or watchdog reset.

**15.2.4 Counter Breakpoint Status Register (CTRS) ASI=0x4b**

This register records the status of either the instruction counter or cycle counter. It will tell whether an interrupt was generated or scan-based debug mode was entered when the cycle or instruction counter overflow occurred.





<b>BCIPL</b>	Breakpoint Interrupt Level. Determines the interrupt level to be used for all breakpoint or counter interrupts. When generated, these interrupts behave the same as external interrupts—they must meet SPARC criteria for PIL and IRL to be seen in the execution stream. This field is cleared at reset.
<b>E_CBK</b>	ESB Strobe on Code Address Breakpoint. Enables the ESB pin to strobe on a code address breakpoint. This bit is cleared at reset.
<b>E_ZIC</b>	ESB strobe on Zero Instruction Count. Enables the ESB pin to strobe on a zero instruction count expiration. This bit is cleared at reset.
<b>E_DBK</b>	ESB strobe on Data Address Breakpoint. Enables the ESB pin to strobe on a data address breakpoint. This bit is cleared at reset.
<b>E_ZCC</b>	ESB strobe on Zero Instruction Count. Enables the ESB pin to strobe on a zero cycle count expiration. This bit is cleared at reset.
<b>I_CBK</b>	Enable Interrupt on Code Breakpoint. Enables generation of an interrupt in response to code breakpoint. This bit is cleared at reset.
<b>I_ZIC</b>	Enable Interrupt on Zero Instruction Count. Enables generation of an interrupt in response to zero instruction count. This bit is cleared at reset.
<b>I_DBK</b>	Enable Interrupt on Data Breakpoint. Enables generation of an interrupt in response to data breakpoint. This bit is cleared at reset.
<b>I_ZCC</b>	Enable interrupt on Zero Cycle Count. Enables generation of an interrupt in response to zero cycle count event. This bit is cleared at reset.

If the event is to generate a scan-based debug request, the associated IEN bit must be cleared. When an interrupt on these events is desired, the associated IEN bit must be set.

## 15.3 External Monitors

The SSP is a highly integrated processor. SuperSPARC can execute code continuously from its internal cache with no external indications. This can make system debugging very difficult.

Several features are provided to reduce these problems. Scan-based debug provides access to the internal state of the machine.

Additional pins have been defined to provide cycle-by-cycle observation of key internal states (PIPE[9:0]). These pins provide information on activity within a clock cycle. The ESB signal has been provided as an external trigger for debug equipment.

Scan-based debug is discussed in more detail in Chapter 22. The PIPE pins and ESB pin are discussed in the following sections.

### 15.3.1 PIPE Pins

The PIPE[9:0] pins provide cycle-by-cycle information on the following events:

- ☐ The number of instructions that complete execution.
- ☐ When branches and memory operations occur (including an indication of taken branches).
- ☐ When floating-point instructions are issued.
- ☐ When the pipeline is being held by either floating-point operations or memory operations.
- ☐ Interrupts and exceptions.

The definitions of these signals are provided below:

PIPE[9]	Active when any valid memory reference occurred in the E0 stage of the previous clock cycle.
PIPE[8]	Active when any valid floating point operation occurred in the E0 stage of the previous clock cycle.
PIPE[7]	Active when any valid control transfer instruction was executed in the E0 stage of the previous clock cycle.
PIPE[6]	Indicates that no instructions were available when the group currently at the WB stage was decoded (D0).
PIPE[5]	Active when the pipeline is being held by the Data Cache. (Generally processing a cache miss).

PIPE[4]	Active when the pipeline is being held by the Floating Point Unit (FPU). (Either queue is full, or dependencies).
PIPE[3]	Indicates that the branch in E0 stage of the previous cycle was taken (1) or not taken (0).
PIPE[2:1]	Indicates the number of instructions in the E0 stage of the current cycle: 00=None, 01=1, 10=2, 11=3.
PIPE[0]	Indicates that an exception or interrupt is being signalled in the current cycle.

### 15.3.2 ESB

The breakpoint registers may be programmed to activate the ESB pin when a breakpoint is detected. This allows an external device to be triggered (oscilloscope, logic analyzer, etc.).

The ESB signal will be asserted when entering scan-based debug mode via a code breakpoint, data address breakpoint, zero instruction count breakpoint, or a zero cycle count breakpoint, and when the appropriate bits in the ACTION register are set. If you are not using scan-based debug mode, assert the ESB signal by setting up an interrupt on either a code address breakpoint, data address breakpoint, zero instruction count breakpoint, or a zero cycle count breakpoint.

Following is an algorithm to enable the ESB signal at a code address or data address breakpoint.

- 1) Execute an STDA of the appropriate value to the ACTION register.

- **Code Address Breakpoint**

To set up a request for ESB assertion on scan-based debug mode entry for a code address breakpoint, the ACTION.E\_CBK bit must be set. The other bits in the ACTION register, except for the ACTION.MIX field, must be cleared. The value of the ACTION.MIX field will depend on your code. To include an interrupt, ACTION.E\_CBK and ACTION.I\_CBK must also be set.

- **Data Address Breakpoint**

To set up a request for ESB assertion on scan-based debug mode entry for a data address breakpoint, the ACTION.E\_DBK bit must be set. The other bits in the ACTION register, except for the ACTION.MIX field, must be cleared. The value of the ACTION.MIX field will depend on your code. To include an interrupt, ACTION.E\_DBK and ACTION.I\_DBK must also be set.

- 2) Set the breakpoint value register to the appropriate breakpoint address value.
- 3) Set the breakpoint mask register to the appropriate mask value.
- 4) Set the breakpoint control register to generate an interrupt or scan-based debug request on the code or data address match. The bits to set will depend on your application. (See Subsection 15.2.1.3 for more information.)
- 5) Clear the breakpoint status register.
- 6) Execute the code until the breakpoint occurs.

Following is an algorithm to enable the ESB signal at a zero instruction count or zero cycle count breakpoint.

- 1) Execute an STDA of the appropriate value to the ACTION register.

- Zero Instruction Count Breakpoint

To set up a request for ESB assertion on scan-based debug mode entry for a zero instruction count breakpoint, the ACTION.E\_ZIC bit must be set. The bits in the ACTION register, except for the ACTION.MIX field, must be cleared. The value of the ACTION.MIX field will depend on your code. To include an interrupt, ACTION.E\_ZIC and ACTION.I\_ZIC must also be set.

- Zero Cycle Count Breakpoint

To set up a request for ESB assertion on scan-based debug mode entry for a zero cycle count breakpoint, the ACTION.E\_ZCC bit must be set. The other bits in the ACTION register, except for the ACTION.MIX field, must be cleared. The value of the ACTION.MIX field will depend on your code. To include an interrupt, ACTION.E\_ZCC and ACTION.I\_ZCC must also be set.

- 2) Set the counter breakpoint value register to the appropriate counter value.
- 3) Set the counter breakpoint control register to enable either the instruction or cycle counter.
- 4) Clear the counter breakpoint status register.
- 5) Execute the code until the breakpoint occurs.

## **MultiCache Controller (MXCC)**

---

---

The MultiCache Controller (MXCC) is an optional external cache controller for SuperSPARC Processors (SSPs). It is employed when a large secondary cache or an interface to a non-MBus system is required.

<b>Topic</b>	<b>Page</b>
16.1 Introduction .....	16-2
16.2 MXCC Block Diagram and Basic Functionality .....	16-8
16.3 Control Space Access .....	16-12
16.4 External Cache (E-Cache) .....	16-14
16.5 MXCC Internal Registers .....	16-21
16.6 MXCC Block Copy Facility .....	16-30
16.7 MXCC Interrupts .....	16-33
16.8 MXCC Error Handling .....	16-36

## 16.1 Introduction

The SuperSPARC MXCC aids in building high-performance systems using the SSP. It provides a number of functions in the system:

- ☐ Selectable bus interface for greater freedom in configuring systems.
- ☐ External (second-level) cache controller and tags for 1-Mbyte (MBus) or up to 2-Mbytes (XBus).
- ☐ Multiprocessor support, including cache consistency.
- ☐ Block copy and block fill controller.
- ☐ Interrupt control functions (mostly in XBus configurations).
- ☐ Cache block prefetching.
- ☐ BootBus (XBus configurations only).
- ☐ Memory model support (XBus configurations).
- ☐ VBus arbitration and control (see Chapter 18).
- ☐ JTAG boundary scan and internal test functions.

### 16.1.1 Configurations

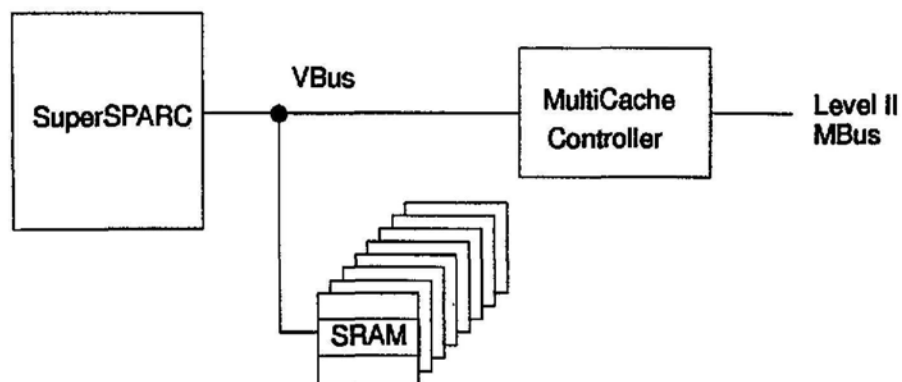
The MXCC offers several degrees of freedom for the design of systems using the SSP, as shown in Table 16-1.

Table 16-1. SuperSPARC Chipset Configurations

Configuration	SSP	MXCC	SRAMs	Bus
Direct MBus	√			MBus
Minimum MBus	√	√		MBus
Minimum XBus	√	√		customer-defined
High-Performance MBus	√	√	√	MBus
High-Performance XBus	√	√	√	customer-defined

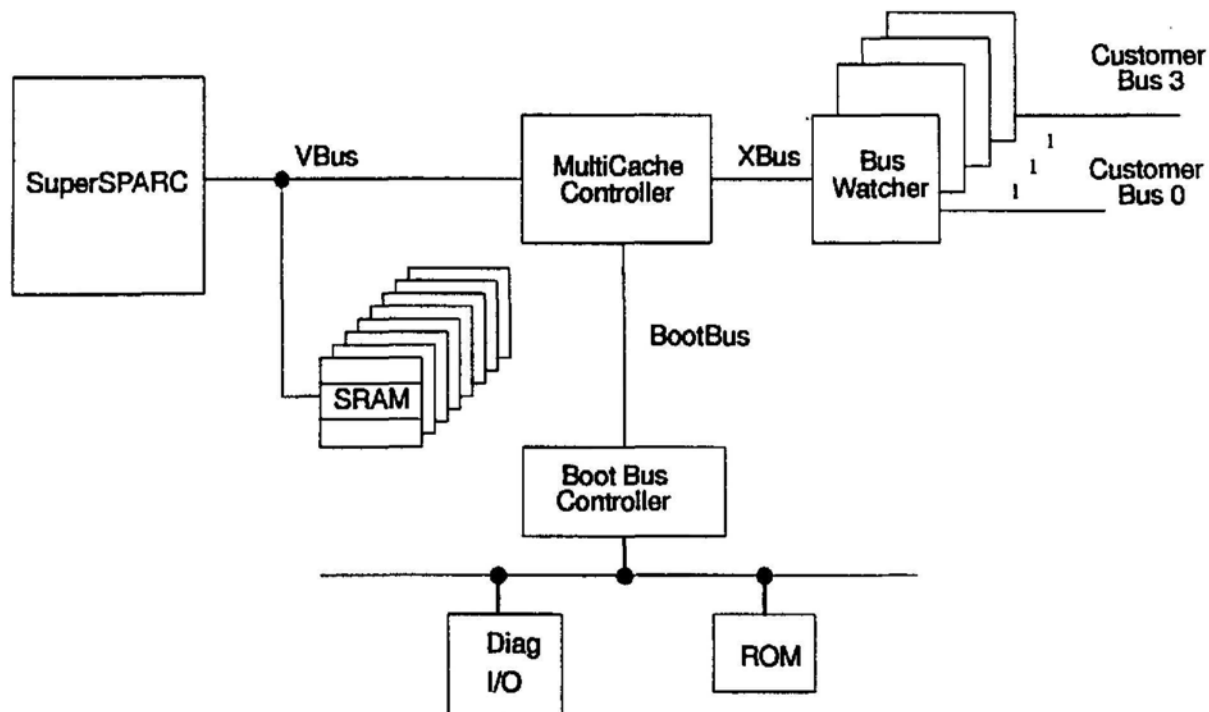
The high-performance MBus system is diagrammed in Figure 16-1. The external cache memory provides significant performance improvement and greatly decreases bus traffic in order to support more processors on an MBus.

Figure 16-1. High-Performance MBus System



If the system bus is not an MBus, the MXCC supports external system bus interfaces in its XBus configurations. The external system bus interface mates the MXCC to a particular system bus. External system bus interfaces are referred to as bus watchers (BWs). Figure 16-2 shows the high-performance XBus system configuration.

Figure 16-2. System With External Bus Watchers





The XBus configurations support up to four external BWs that can be used with several system buses to increase available bandwidth between processors and memory.

External cache RAM may not be needed in every application. Configurations without external cache are listed as minimum configurations in Table 16-1.

The external cache (E-cache) for SuperSPARC is a large secondary physically addressed cache controlled by the MXCC. The MXCC stores the tags for E-cache internally and uses external synchronous SRAM chips for data storage. Each SRAM receives the SSP's clock, PCLK.

### ***Synchronous SRAM***

Synchronous SRAMs have registers on each input and output. This allows pipelined operation. An address is presented to an SRAM before the active clock edge; the address is registered in the SRAM at the clock edge. The SRAM reads out the addressed location before the next active clock edge, and the result is stored in an output register at the edge. New addresses can be supplied at each clock edge. New outputs appear after two clock periods of delay. Writing works similarly. Address, data, output enable, and write-enable are registered on the active clock edge and stored into the internal array during the subsequent clock period.

The E-cache is organized as a direct-mapped cache with a normal size of 1 Mbyte. This configuration is implemented with eight 128Kx8 or 128Kx9 synchronous SRAMs. The 128K x 9 SRAMs are needed to implement byte parity on the E-cache data storage. Parity is directly supported by both SuperSPARC and the MXCC.

### ***Bus Connection***

The MXCC supports a direct connection to a Level 2 MBus (see Chapter 17). Alternatively, the XBus may be selected (see Chapter 19). XBus connects MXCC to an external BW that interfaces to different types of system buses. XBus supports up to four BWs.

The MXCC is configured for either the MBus or the XBus operation with the MBSEL pin. A number of other parameters are altered by the choice of bus configuration. These are summarized in Table 16-2.

Table 16–2. Configuration Changes With MBSEL Pin

Feature	MBus Interface MBSEL=H	XBus Interface MBSEL=L
Sub-block Size	32 bytes	64 bytes
Block Size	128 B	256 B
Cache Size	1 MB	0.5, 1, 2 MB
Boot-bus	not available	available
Interrupts	from pins	from XBus packets

### 16.1.2 Cache Tags and Control

The MXCC contains tags for an E-cache that acts as a second-level cache in the system. The MXCC also controls accesses to the cache: sequencing accesses, handling E-cache misses, selecting blocks for replacement, and handling snoop requests from the bus interface. The cache tags can support a direct-mapped external cache of 1 Mbyte in an MBus configuration or up to 2 Mbyte in an XBus configuration.

The internal caches in the SSP obey *inclusion* with respect to the E-cache. Any data in the internal cache must also be present in the E-cache. Since only the E-cache tags need be examined to determine whether a snoop hits in any of the three caches, bus snooping is greatly simplified. In order to enforce inclusion, whenever data is removed from the E-cache due to either an invalidation caused by a snoop hit or a block replacement, the same block must also be invalidated in the internal instruction cache and the internal data cache wherever it may reside.

When the MXCC is used in the MBus configuration, it supports either no E-cache or 1 Mbyte of E-cache. When the MXCC is used in an XBus configuration, it supports a variety of E-cache sizes: none, 512 Kbyte, 1 Mbyte, or 2 Mbyte.

The E-cache is blocked and sub-blocked. The sub-block size is 32 bytes in MBus configurations and 64 bytes in XBus configurations. There are four sub-blocks per block. Block size is 128 bytes in MBus configurations and 256 bytes in XBus configurations.

The MXCC supports pipelined access to the E-cache from the SSP. The peak data rate that can be achieved is one double-word (DW) every cycle for either read and write.

The MXCC can handle one read miss and one write miss at any given time. In addition, some system buses can have more than one access in progress (XBus configurations).

The E-cache tags are accessible for diagnostic access through control space. The E-cache data is also accessible through control space, as are the MXCC control registers that control the behavior of the cache controller and bus interfaces.

### 16.1.3 Multiprocessor Support

The MXCC has extensive multiprocessor support. In MBus configurations, the MXCC contains all functions of a snooping-coherent cache and bus master. See Chapter 17 for more information on the MXCC's behavior on MBus.

In XBus configurations, the MXCC and BWs cooperate to support cache coherence and high-performance multiprocessing in the system environment. See Chapter 19 for more information on XBus cache consistency and protocols.

### 16.1.4 Block Copy and Block Fill

The MXCC can perform high-performance block copy and block fill operations in cooperation with the SSP. The MXCC's buffer can be loaded with a high-performance block read on the system bus and stored with a high-performance block write. The buffer holds a single-cache sub-block, 32 bytes in MBus configurations, and 64 bytes in XBus configurations.

Sequences of load block and store block operations can be initiated in rapid succession by the SSP to perform a block copy at the system bus's burst transfer rate.

Block fill can also be performed by loading the buffer with the fill pattern and by performing a series of block writes.

### 16.1.5 Interrupt Control Functions

The MXCC directly drives the IRL[3:0] pins of the SSP.

In MBus configurations, the system presents interrupt requests to the MXCC/SSP pair via MXCC's IRL[3:0] pins. No additional interrupts are generated internally inside MXCC.

In XBus configurations, the system presents interrupt requests to the MXCC/SSP pair via XBus packets. In addition, the MXCC generates level-15 interrupts to the SSP to report asynchronous errors. See Section 16.7 for more details.

### **16.1.6 Cache Block Prefetching**

The MXCC supports prefetching of sub-blocks into the E-cache in order to reduce the effective memory latency during sequential accesses to memory. Prefetching is triggered when the MXCC services a miss from the SSP's internal instruction or data cache and the next (sequential) sub-block within the E-cache block is not in the E-cache.

Prefetching is controlled by a bit in the CCCR. See the description of CCCR.PF in Subsection 16.5.3.

### **16.1.7 Boot Bus**

In XBus configurations, the MXCC supports an eight-bit bus for communication with local peripherals and boot ROM. See Chapter 20 for more details.

### **16.1.8 Memory Model Support**

The MXCC, if in an XBus configuration, can have several outstanding transactions in the system. The MXCC keeps track of outstanding transactions. To aid in supporting the PSO memory model, MXCC asserts **PEND** whenever it has a store operation pending in the system.

See Section 8.7.

### **16.1.9 VBus Arbitration and Control**

The MXCC acts as the VBus arbiter when used with the SSP. MXCC controls the **FGNT** and **WGNT** signals in order to control access to the VBus by the SSP.

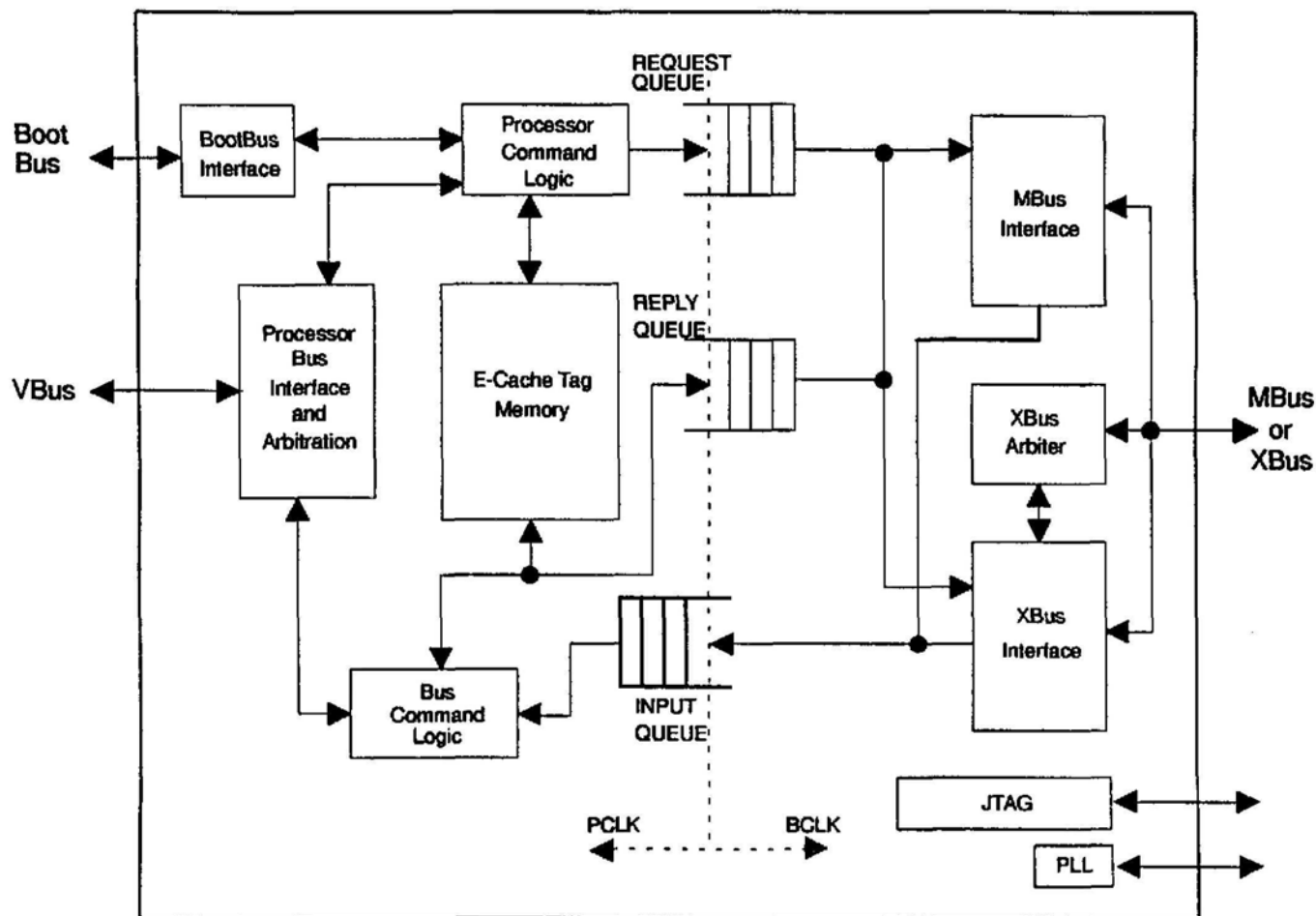
See Chapter 18.

### **16.1.10 JTAG Boundary Scan and Test**

The MXCC has full boundary scan accessible by the JTAG test access port. In addition, internal logic is testable via the JTAG port. See Section 21.6, for more details.

## 16.2 MXCC Block Diagram and Basic Functionality

Figure 16–3. MXCC Block Diagram



The MXCC can be partitioned into five large functional blocks:

- ☐ **Cache Controller Core.**  
This includes the E-cache tag memory, the processor command logic, and the bus command logic.
- ☐ **SSP Interface.**  
This includes the processor bus interface and arbitration logic.
- ☐ **MBus/XBus Interface.**  
This includes the MBus interface, the XBus interface, and the XBus arbiter.

☐ **Queues and Synchronizers.**

This includes the input and output queues (input queue and the request and reply queues) and their synchronizers.

☐ **BootBus Interface.**

There are two clock domains on the MXCC. These are delineated by the vertical dotted line in Figure 16-3. The side connected to the SSP runs from the processor clock (PCLK), while the side connected to the external bus runs at the bus clock (BCLK). Special provisions allow for synchronous operation when the two clocks are the same; see Chapter 23 for more details.

### 16.2.1 Cache Controller Core

The cache controller core is comprised of the E-cache tag memory, the processor command logic, and the bus command logic. The E-cache tag memory keeps track of the usage of the external cache. The tag memory is organized as  $8K \times 33$ -bit memory and supports 8K blocks, with four sub-blocks per block; 33 bits of information are kept for each block. These record the address tag (up to 17 bits) and four status bits for each sub-block (16 bits). In MBus configurations, sub-blocks are 32 bytes; 1 Mbyte of E-cache is supported. In XBus configurations, sub-blocks are 64 bytes; up to 2 Mbyte can be supported.

In the MBus configurations, the E-cache tag memory is used for both E-cache access and bus snooping. When used with XBus, the tag memory is used only for E-cache accesses. Snooping in the XBus configuration is accomplished by a bus watcher that provides an interface between XBus and a system bus.

The processor command logic is a finite state machine that handles incoming processor commands. If a processor requires access to either MBus or XBus, the command logic generates a bus command through the request queue. The command logic also deals with acknowledgements of bus requests and, in the case of reads, delivers the requested data.

The bus command logic handles all the requests from the bus in the input queue, then places replies to them in the output queue. Together with the MBus interface, it implements the MBus cache-consistency protocol. The bus command logic, when combined with a suitable bus watcher on XBus, can provide other cache-consistency protocols. In response to an external bus request, the command logic may access E-cache tag memory, the E-cache RAMs, and/or the processor.

### **16.2.2 SSP Interface**

The MXCC interfaces to the SSP via VBus (see Chapter 18). MXCC's SSP interface provides this interface and consists of the processor bus interface and arbitration logic.

The processor bus interface provides the bus command logic with the illusion of a free SSP. Thus the bus command logic can issue writes to E-cache with no arbitration. The interface logic uses a buffer to store up to nine cycles of VBus accesses from the bus command logic. Since the interface logic buffers VBus accesses, VBus arbitration is hidden from the bus command logic.

The processor bus interface logic latches in all the signals from the processor (except for a few control strobes) before using them in the MXCC. This logic also latches all the output signals before driving them out.

The processor arbitration logic is responsible for arbitrating the usage of VBus among the SSP, the processor command logic, and the bus command logic. Read and write cycles are arbitrated separately on VBus via separate read grant and write grant lines to the processor.

### **16.2.3 MBus/XBus Interface**

The MBus/XBus interface is composed of the MBus interface, the XBus interface, and the XBus arbiter. MXCC operates either the MBus interface or the XBus interface, as selected by the MBSEL pin. When MBSEL is high, the MBus interface is selected.

The MBus interface handles the MXCC's interface to a system bus using the MBus level 2 protocol (see Chapter 17). The MBus interface receives commands generated by the processor command logic in the request queue and initiates MBus transactions. The replies from these transactions are placed in the input queue for the bus command logic.

All requests on the MBus, except for non-cacheable accesses, are put into the input queue to send to the bus command logic for snooping. For non-cacheable accesses, the MBus interface decodes the address and places only accesses to this module in the input queue. The MBus interface takes the reply to a pending MBus request, either one addressed to this module or a snoop request, from the reply queue and delivers it to the MBus.

The XBus Interface handles the MXCC's side of the XBus protocol (see Chapter 19). The XBus Interface takes a command generated by the processor command logic from the request queue and initiates an XBus transaction. When a reply packet is returned on the bus, the interface puts it into the input queue. VBus transactions initiated by the processor are processed by the processor command logic and sometimes generate a request by placing an entry in the request queue.

All requests on XBus addressed to the MXCC are put into the input queue to send to the bus command logic. The XBus interface takes the reply of a pending XBus request from the reply queue and delivers it to the XBus.

The XBus arbiter controls access to the XBus among the MXCC and the BWs. Up to four BWs are supported by the XBus arbiter. The bus is granted according to arrival order and priority of bus request signals from the BWs. (See Section 19.10.)

#### **16.2.4 Queues and Synchronizers**

The input queue, request queue, and reply queue are first-in, first out (FIFO) queues used to communicate between the two clock domains. They are implemented with dual-port register files. The input queue is a  $16 \times 73$ -bit buffer, while the request queue is made up of two  $10 \times 65$ -bit buffers. The reply queue consists of three  $10 \times 65$ -bit buffers. The input queue is read with PCLK and written with BCLK, while the Request and Reply Queues are read with BCLK and written with PCLK.

Control strobes are sent between the two domains through synchronizers. The synchronizers can be defeated for synchronous operation, where PCLK and BCLK are the same (see Chapter 23).

#### **16.2.5 Boot Bus Interface**

The boot bus interface handles access to boot bus. The BootBus interface implements the address and data multiplexing functions to the boot bus, along with the automatic polling of interrupts (see Chapter 20).



### 16.3 Control Space Access

The SSP's ASI 0x02 allows access to the information in an external device—in this case, the MXCC—via VBus's control space. These accesses are non-cacheable, regardless of MCNTLAC indication. SuperSPARC references to this control space may be any size and are properly aligned on the VBus. Any fault is reported as a `data_access_exception`.

The MXCC's control space occupies different addresses, depending on whether the MXCC is in an MBus or XBus configuration and whether it is addressed from the system bus or the processor. Table 16-3 shows control space base addresses for processor and bus access from MBus and XBus configurations.

Table 16-3. MXCC Control Space Base Addresses

Access From	Base Address
Processor (VBus)	CSA = 0 ADDR[35:25] ignored ADDR[24] = 1
MBus	MAD[35:24] = 0xFFn n is MID[3:0]
XBus	PA[35:25] ignored PA[24] = 1

The E-cache data, E-cache tags, and the MXCC's registers are all accessed via control space. The address within control space for each of these is shown in Table 16-4.

Table 16-4. MXCC Control Space Address Decoding

Access	Starting Offset	Decode	Reference
E-cache Data	0x000000	PA[23] = 0 PA[22:20] reserved	Figure 16-5
E-cache Tags	0x800000	PA[23:22] = 10 PA[21:20] reserved	Figure 16-6
MXCC Registers	0xC00000	PA[23:22] = 11 PA[21:20] reserved	Figure 16-8

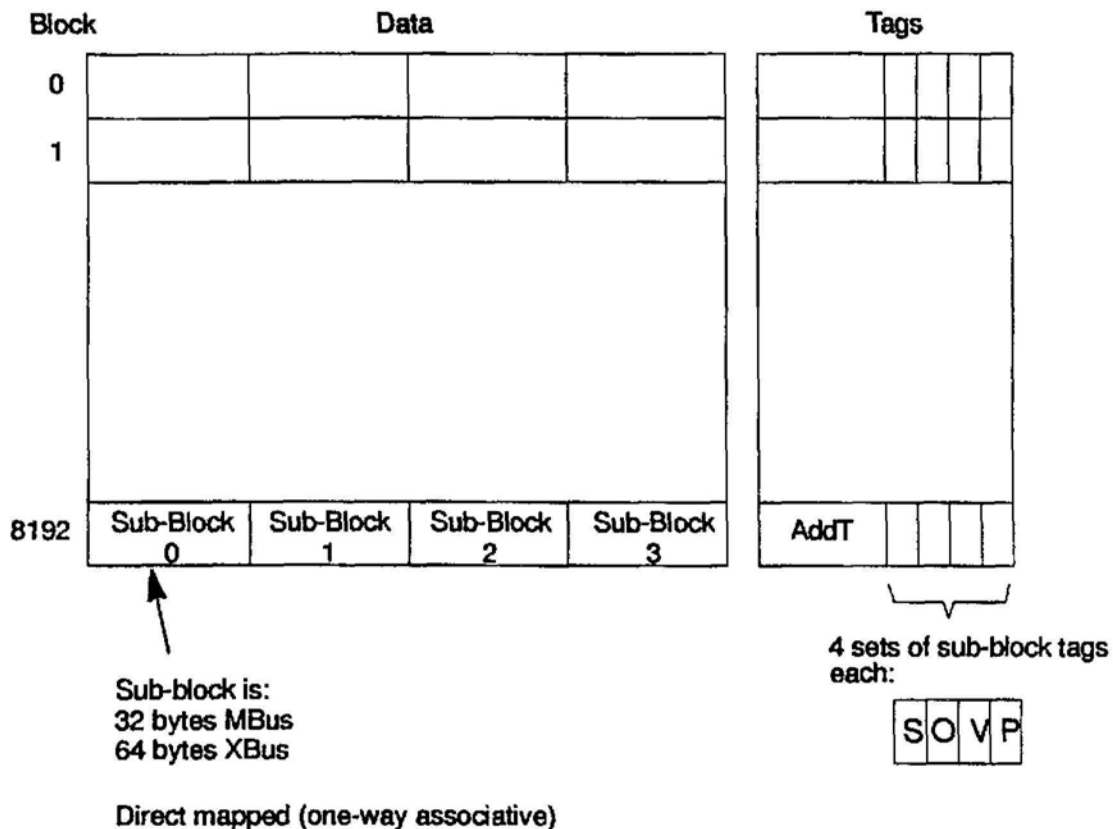
PA = Physical Address

Any illegal access by the processor to the MXCC control space generates a timeout (TO) bus error. Illegal access includes any illegal address, atomic load-store to any MXCC register, or writes to read-only registers. Illegal access to the MXCC from the MBus or XBus is ignored, and it is the system's responsibility to generate timeout responses.

## 16.4 External Cache (E-Cache)

The MXCC controls SuperSPARC's E-cache. E-cache is a direct-mapped cache; there is a single cache location at which any particular byte of the physical address space can reside in cache. Many different bytes have the same place in the cache. E-cache is a copy-back cache; writes into the E-cache do not propagate to main memory until the cache block is replaced. Figure 16-4 shows the organization of the external cache memory.

Figure 16-4. E-Cache Organization



With the MXCC on MBus, the external cache can store 1 Mbyte. With MXCC on XBus, the E-cache can be configured to store 2 Mbytes, 1 Mbyte, or 512 Kbytes. See Table 16-5 for CCCR bit settings for various E-cache sizes. The width and position of many fields for E-cache access depend on the configuration of the E-cache.

Table 16-5. MXCC Effective Size of E-Cache

CS	HC	Effective E-cache Size	
		XBus	MBus
0	0	1MB	1MB
0	1	512KB	1MB
1	0	2MB	1MB
1	1	reserved	1MB

The E-cache is a unified cache; it combines instructions and data in a single cache and supplies both data and instructions to the processor. The MXCC maintains the inclusion property on SuperSPARC's on-chip caches with respect to the E-cache, which means that there will be no cache block in either of SuperSPARC's internal caches that is not also in E-cache. Inclusion is maintained by allocating a block in the E-cache for any cacheable data accessed by the processor, and by invalidating the corresponding blocks in the internal caches whenever an E-cache block is invalidated.

Each cache block contains four sub-blocks. The sub-block size is 32 bytes in MBus configurations and 64 bytes in XBus configurations. Block size is, therefore, 128 bytes in MBus systems and 256 bytes in XBus configurations. The block size remains the same across the various XBus cache sizes.

In MBus configurations, non-cacheable accesses by the processor generate Level-1 MBus requests. In XBus configurations, non-cacheable accesses are processed similarly to a read miss or a shared write, but the E-cache data is not updated, and no sub-block is ever replaced.

The E-cache can be accessed directly from SuperSPARC in the MXCC's control space for the purposes of initialization, testing, and diagnostics.

#### 16.4.1 E-Cache Data Access

E-cache data memory can be accessed directly in the MXCC control space for testing, initialization, and diagnostics. The E-cache data can be accessed with a byte, half-word, word, or double-word access. The format of an address within control space depends on MBSEL and on the cache size (CS) and half cache (HC) bits in the MXCC control register (CCCR). The CCCR.CS and CCCR.HC bits are ignored when MXCC is used on the MBus. The field widths are shown in Table 16-6; the field positions are shown in Table 16-7.

Table 16–6. E-Cache Data Address Field Widths (in bits)

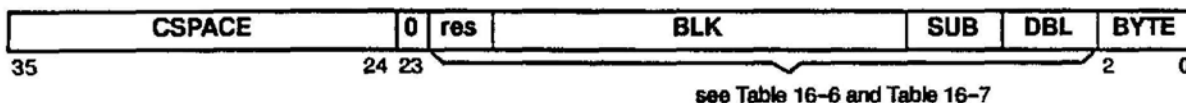
Field	Description	MBus	XBus		
		1MB	512KB	1MB	2MB
BYTE	Byte	3	3	3	3
DBL	Double Word	2	3	3	3
SUB	Sub-block	2	2	2	2
BLK	Block	13	11	12	13
res	Reserved	3	4	3	2

Table 16–7. E-Cache Data Address Field Positions

Field	Description	MBus	XBus		
		1MB	512KB	1MB	2MB
BYTE	Byte	[2:0]	[2:0]	[2:0]	[2:0]
DBL	Double Word	[4:3]	[5:3]	[5:3]	[5:3]
SUB	Sub-block	[6:5]	[7:6]	[7:6]	[7:6]
BLK	Block	[19:7]	[18:8]	[19:8]	[20:8]
res	Reserved	[22:20]	[22:19]	[22:20]	[22:21]

The address of a byte in E-cache is shown in Figure 16–5. The field widths from Table 16–6 and field positions from Table 16–7 are needed to complete the address description for a particular E-cache configuration.

Figure 16–5. Addressing E-Cache Data



- CSPACE** Control Space Base Address. Base address of the MXCC control space as shown in Table 16–3.
- res** Reserved. These bits are ignored and should be zero. The width and position of this field vary with E-cache configuration and are shown in Table 16–6 and Table 16–7, respectively.
- BLK** Block Number. This selects the block of the E-cache data to access. The width and position for this field vary with E-cache configuration and are shown in Table 16–6 and Table 16–7, respectively.

<b>SUB</b>	Sub-block Number. This selects a sub-block of the E-cache data to access. The width and position of this field vary with the E-cache configuration and are shown in Table 16-6 and Table 16-7, respectively.
<b>DBL</b>	Double Word. This selects a double word in an E-cache sub-block to access. The width and position of this field vary with the E-cache configuration and are shown in Table 16-6 and Table 16-7, respectively.
<b>BYTE</b>	Byte. This selects a byte of an E-cache sub-block to access.

#### 16.4.2 External Cache Tags

The external cache tags can be accessed in the control space for the purposes of testing, initialization, and diagnostics. The address format is similar to the E-cache data access address format. Only the block number field is used for E-cache tag access.

The size and position of several fields in the address vary according to the E-cache configuration. The size and position of these fields is shown in Table 16-8 and Table 16-9, respectively.

Table 16-8. E-Cache Tag Address Field Widths (in bits)

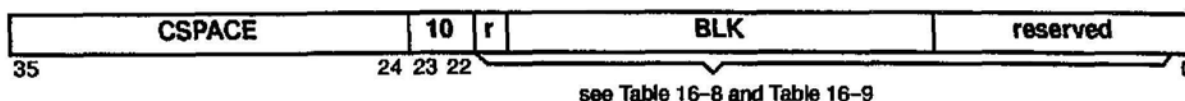
Field	Description	MBus	XBus			
		1MB	512KB	1MB	2MB	
reserved	Reserved	7	8	8	8	
BLK	Block	13	11	12	13	
r	Reserved	2	3	2	1	

Table 16-9. E-Cache Tag Address Field Positions

Field	Description	MBus	XBus			
		1MB	512KB	1MB	2MB	
reserved	Reserved	[6:0]	[7:0]	[7:0]	[7:0]	
BLK	Block	[19:7]	[18:8]	[19:8]	[20:8]	
r	Reserved	[21:20]	[21:19]	[21:20]	[21]	

The format of an E-cache tag address is shown in Figure 16-6.

Figure 16–6. Addressing E-cache Tags



<b>CSPACE</b>	Control Space Base Address. Base address of the MXCC control space is shown in Table 16–3.
<b>r, reserved</b>	Reserved. These fields are ignored and should be zero. The widths and positions of these fields vary with the E-cache configuration and are shown in Table 16–8 and Table 16–9, respectively.
<b>BLK</b>	Block Number. This field selects the tag of one E-Cache block to access. The width and position of this field vary according to the E-cache configuration and are shown in Table 16–8 and Table 16–9, respectively.

The format of an E-cache tag entry is shown in Figure 16–7. E-cache tag entries are of DW length. Accesses to E-cache tags are assumed to be DW length; the VBus size bits (SIZE[1:0]) are ignored, and DW data is always supplied.

The field sizes and positions of the res1 and AddT fields depend on the E-cache size selected. The sizes of these fields are shown in Table 16–10, and their positions are shown in Table 16–11.

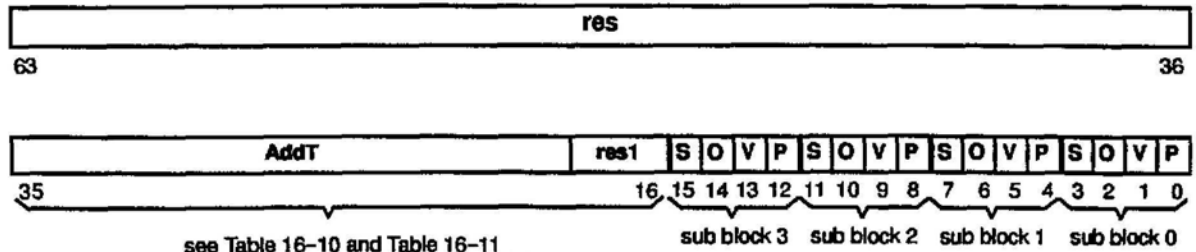
Table 16–10. E-Cache Tag Field Widths (in bits)

Field	Description	MBus	XBus		
		1MB	512KB	1MB	2MB
res1	Reserved	4	3	4	5
AddT	Address Tag	16	17	16	15

Table 16–11. E-Cache Tag Field Positions

Field	Description	MBus	XBus		
		1MB	512KB	1MB	2MB
res1	Reserved	[19:16]	[18:16]	[19:16]	[20:16]
AddT	Address Tag	[35:20]	[35:19]	[35:20]	[35:21]

Figure 16-7. External Cache Tag Format



see Table 16-10 and Table 16-11

The various bit fields are:

<b>res</b>	Reserved. Read as 0 and ignored on write.
<b>AddT</b>	Address Tag. This is the high-order part of the address that matches this E-cache block. The middle part of the address selects a block of the cache according to the direct mapped scheme and is implied in the address comparisons. The lower-order portion of the address is used to select the sub-block, double word, and byte to access. E-cache lookups compare the corresponding field of the physical address with the AddT field of the E-cache tags for the E-cache block selected by the physical address. If the AddT field matches and the addressed sub-block is valid, the access is a cache hit.  The size and position of this field vary depending on the E-cache configuration and are shown in Table 16-10 and Table 16-11.
<b>res1</b>	Reserved. Read as 0 and ignored on write. The size and position of this field vary depending on the E-cache configuration and are shown in Table 16-10 and Table 16-11.
<b>S</b>	Shared. When set, the corresponding sub-block is shared. When clear, the sub-block is held exclusively, if valid. There are four S bits, one for each sub-block.
<b>O</b>	Owner. When set, this processor has a dirty copy of the data in this sub-block. The data will be supplied by this cache to the bus if another processor requests it from memory. If this block is replaced, this sub-block must be copied back to memory. If clear, main memory or another cache is the owner of this data. There are four O bits, one for each sub-block.



## External Cache (E-Cache)

---

- V** Valid. When set, this sub-block contains valid data and can be used. When clear, this sub-block does not contain valid data. Attempts to access an invalid sub-block will cause a cache miss. There are four V bits, one for each of the four sub-blocks in a block.
- P** Pending. When set, a VBus operation is pending on this sub-block. This bit is set automatically when MXCC has issued an operation on the system bus for this sub-block and is cleared automatically when the operation completes. There are four P bits, one for each of the four sub-blocks.

## 16.5 MXCC Internal Registers

The MXCC contains a number of registers that control the operation of E-cache, bus, and other functions or that sense the status of MXCC functions. These registers are accessible in the MXCC's control space (see Table 16-3).

The address format for accessing MXCC registers is shown in Figure 16-8.

Figure 16-8. Addressing MXCC Registers

CSPACE	1 1	reserved	SEL	r	DBL	zero
35	24 23 22 21		12 11	8 7 6 5	3 2	0

<b>CSPACE</b>	Control Space Base Address. Base address of the MXCC's control space as shown in Table 16-3.
<b>reserved, r</b>	Reserved. This field is ignored and should be zero.
<b>SEL</b>	Select Field. This field selects one of the MXCC's internal registers. Table 16-12 shows the selectors for the various MXCC registers.
<b>DBL</b>	Double Word. This field is ignored except when accessing the stream data register (SEL = 0x0).
<b>zero</b>	This field must be zero.

The SEL field is used to select one of the MXCC internal registers. The correspondence of SEL values to registers is shown in Table 16-12. This table also shows an abbreviated register name, the width of the register, and the section in this chapter where the register is described in detail.

Table 16–12. MXCC Register Selectors

SEL Field	Register	Abbrev	Width	Section
0x0	Stream Data		64 bits	16.6.1
0x1	Stream Source		64 bits	16.6.2
0x2	Stream Destination		64 bits	16.6.3
0x3	Reference/Miss Count		64 bits	16.5.1
0x4	Interrupt Pending		16 bits	16.7.1
0x5	Interrupt Mask		16 bits	16.7.2
0x6	Interrupt Pending Clear		16 bits	16.7.3
0x7	Interrupt Generation		32 bits	16.7.4
0x8	BIST		32 bits	16.5.2
0x9	reserved			
0xA	MXCC Control	CCCR	32 bits	16.5.3
0xB	MXCC Status	CCSR	64 bits	16.5.4
0xC	Reset		32 bits	16.5.5
0xD	reserved			
0xE	Error	CCER	64 bits	16.8.1
0xF	MBus Port		32 bits	16.5.6

Accesses to registers are assumed to be double-word accesses. On an access from VBus, the size bits, SIZE[1:0], are ignored. During read accesses, the contents of a 32-bit register are driven onto DATA[31:0], and the contents of a 16-bit register are driven onto DATA[15:0]. The 64-bit registers drive all data bits, DATA[63:0]. The unused bits are driven low on read and ignored on write. An atomic load-store operation to any of the registers is not supported. A TO bus error is reported when the processor issues an atomic load-store to any MXCC register.

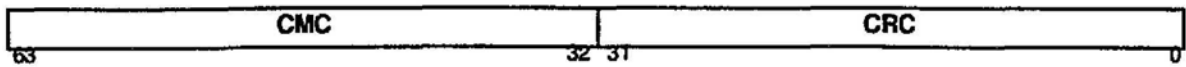
Similarly, system bus access assumes 64-bit access. On reads, registers whose widths are less than 64 bits return unpredictable data, except in fields defined for that register. On writes, the unused bits are ignored.

References to reserved register selectors (0x9 or 0xD) are ignored from the system bus and reported as a TO bus error when issued from the processor.

### 16.5.1 Reference/Miss Count Register

The reference/miss count register can be used to calculate the hit ratio of the E-cache. The register has two fields. Both the CMC and CRC are 32-bit counters, and both may be read and written; however, the two fields must be accessed together as a single DW.

Figure 16–9. MXCC Reference / Miss Count Register



**CMC** Cache Miss Counter. This field counts E-cache misses.

**CRC** Cache Reference Counter. This field counts E-cache references, both hits and misses.

When the high-order bit of the CRC becomes 1, both CMC and CRC are frozen until the bit is cleared by software. CMC will wrap around to zero if it reaches its maximum count. CRC will freeze at its maximum count and therefore cannot wrap around.

Every access is counted in CRC and those that miss the E-cache are also counted in CMC. In normal usage CMC never exceeds CRC, since software normally resets both fields to zero.

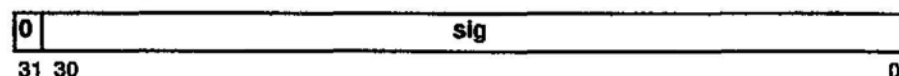
When the RC field in the CCCR is set, only read accesses will be counted in CRC and only read misses will be counted in CMC. When CCCR.RC is clear, both read and write accesses are counted in CRC and misses in CMC.

### 16.5.2 MXCC Built-In Self-Test (BIST) Register

The built-in self-test (BIST) register is a 32-bit register. When the BIST register is written, a BIST is initiated. When read, it returns the signature of the last BIST operation performed. If no BIST operation has been performed, the value is indeterminate. The signature in the BIST register after running BIST should be compared to the known good signature for this revision of the device. Good signatures for current and previous revisions of the MXCC may be found in the data sheet. The value stored in the BIST register cannot be altered by writes, since writing it initiates a BIST operation.

The format for reading of the BIST register is shown in Figure 16–10. Data is ignored on writes to the BIST register.

Figure 16–10. MXCC BIST Register Format



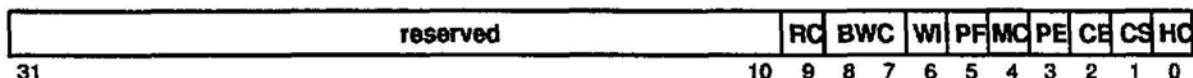
**slg** Signature. This field contains the signature from the last BIST operation performed. If no BIST operation has been performed, the value is indeterminate.

This register is writable from VBus only. When the SSP writes to the BIST register, the MXCC immediately deasserts **RGRT** and **WGRT**. All pending operations are allowed to complete, after which an internal BIST operation begins. During the BIST operation, all bi-directional outputs are high-impedance, and all uni-directional outputs are deasserted. During the BIST operation, the MXCC does not respond to any inputs from the system bus (MBus or XBus). At the end of the BIST operation, the MXCC resets itself and asserts both **RGRT** and **WGRT**.

### 16.5.3 MXCC Control Register (CCCR)

The CCCR contains many of the flags that control the operation of the MXCC. This is a 32-bit read/write register. All flags are cleared at system reset (**RSTIN**). The fields of the control register are shown in Figure 16–11.

Figure 16–11. MXCC Control Register (CCCR)



The fields of the MXCC Control Register are:

**reserved** Read as zero and ignored on write.

**RC** Read reference count. When this bit is set, only read references are counted in the Reference/Miss Count Register. When clear, both read and write accesses are counted.

**BWC**

Bus watcher count. This specifies the number of bus watchers connected to the XBus (see Table 16-13). Notice that the MXCC supports 1, 2, or 4 BWs, but not 3. This field is ignored in the MBus configuration. In the XBus configuration, this value must be set by software before any system bus command is issued.

Table 16-13. MXCC Encoding of BWC Field

BWC	Number of BWs
00	1
01	2
10	reserved
11	4

**WI**

Write invalidate. When this bit is set, a write to shared data (except for atomic load-store operations) invalidates any copies in the other caches. If there is a pending operation on the sub-block (as indicated by the sub-block's P bit), invalidation in other caches is not performed. This bit is used only in the XBus configuration and is ignored in the MBus configuration.

**PF**

Prefetch enable. When this bit is set, a prefetch is triggered on every burst read access (e.g., fetch of a cache block into the SSP's internal instruction or data cache) whose next sequential sub-block is not in E-cache, although prefetch will not cross a block boundary. Only one outstanding prefetch is allowed at any one time. Individual prefetches may occasionally be skipped if there is temporary congestion of resources within the MXCC.

**MC**

Multiple command enable. When this bit is cleared, the MXCC will not accept a new operation from the SSP until it completes any previously initiated operation. When this bit is set, the MXCC may issue multiple commands to the bus or into the output FIFO without waiting for the completion of previous commands.

<b>PE</b>	Parity enable. When set, even parity is generated and checked for each byte on VBus. When the bit is cleared, parity checking is disabled, and odd parity is generated. By generating odd parity when parity checking is disabled, parity errors can be injected into the E-cache for the purpose of verifying parity-detection circuits on VBus.
<b>CE</b>	E-cache enable. E-cache is enabled when this bit is set and disabled when it is clear. When the E-cache is disabled, normal cacheable accesses by the processor do not read or write the E-cache data, but diagnostic access can still be performed via control space.
<b>CS</b>	E-cache Size. When this bit is set, the MXCC supports a 2-Mbyte E-cache. When this bit is clear, 1-Mbyte E-cache is supported. This bit is ignored in MBus configurations. The effective size of E-cache is determined by CS and HC, as shown in Table 16-5.
<b>HC</b>	Half cache. When this bit is set, only the lower half of the E-cache is used. When this bit is cleared, all of the E-cache is used. This bit is ignored in MBus configurations. The effective size of E-cache is determined by CS and HC, as shown in Table 16-5.

#### 16.5.4 MXCC Status Register (CCSR)

The MXCC status register (CCSR) is a 64-bit register that shows the internal state of the MXCC. The status register is readable and writable and is accessible through the JTAG port. The register is writable for diagnostic purposes. Writing this register during normal operation will cause unpredictable results. It is cleared at system reset (RSTIN). The format of CCSR is shown in Figure 16-12.

Figure 16-12. MXCC Status Register (CCSR)

reserved	SXP	sm	NCSID	NCSPA	NCSPC	SPC	BC	WP	RP	PP					
63	40	39	38	37	36	35	12	11	8	7	4	3	2	1	0

The fields of the MXCC Status Register are:

**reserved**                      Reserved. Read as zero and ignored on write.

<b>SXP</b>	Store exception pending. This bit is set when a store operation from the SSP is to be retried, but the bus transaction it triggered generates an exception. The exception is not passed to the processor until the store is retried. When the store is retried, the MXCC signals the error code to the SSP in the usual way with <b>MEXC</b> , <b>WRDY</b> , and <b>RETRY</b> and clears the SXP bit. A store operation is retried on a store miss and on a shared write in the MBus configuration.
<b>sm</b>	Synchronous mode. This bit shows the status of the <b>SYNC</b> pin and is read-only. This bit reads as a 1 if <b>SYNC</b> is asserted (low).
<b>NCSID</b>	Non-cacheable store bus watcher ID. This field indicates the BW ID of the BW with any pending non-cacheable stores. It is used only in the XBus configuration. Multiple non-cacheable stores may be pending only within the same page and via the same BW. There is no such restriction in the MBus configuration, where the MXCC can accept up to two non-cacheable stores into the request queue.
<b>NCSPA</b>	Non-cacheable store page address. This field contains the page address of any pending non-cacheable stores. Multiple non-cacheable stores may be pending only within the same page and BW. There is no such restriction in the MBus configuration, where the MXCC can accept up to two non-cacheable stores into the request queue.
<b>NCSPC</b>	Non-cacheable store pending count. This four-bit counter keeps track of the number of pending non-cacheable store operations. The <b>PEND</b> pin will be asserted if NCSPC is not zero. The SSP uses the <b>PEND</b> signal for memory model support; see Section 8.7 for more details.
<b>SPC</b>	Store pending count. This four-bit counter keeps track of the number of pending cacheable store operations. The <b>PEND</b> pin will be asserted if SPC is not zero. The SSP uses the <b>PEND</b> signal for memory model support; see Section 8.7 for more details.

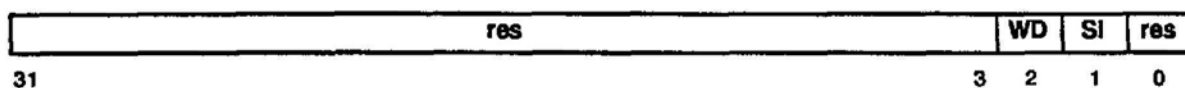


<b>BC</b>	Boot communication. This bit is readable and writable in the MXCC status register and from JTAG scan. The bit can be used to communicate between software running on the SSP and a JTAG scan controller system.
<b>WP</b>	Write miss pending. Set when any write miss has been issued to the bus interface and has not yet completed.
<b>RP</b>	Read pending. Set when any read (except for a prefetch read) has been issued to the bus interface and has not yet completed.
<b>PP</b>	Prefetch pending. Set when any prefetch read has been issued to the bus interface but has not yet completed.

### 16.5.5 MXCC Reset Register

The reset register is a 32-bit register that shows the type of reset that last occurred; see Chapter 13 for more details. The reset register is readable, writable, and JTAG-scannable. The format of the reset register is shown in Figure 16–13.

Figure 16–13. Reset Register



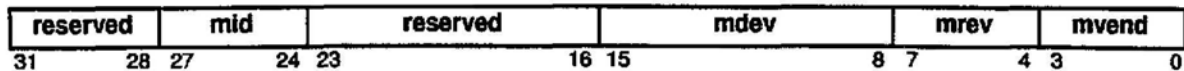
The fields of the Reset Register are:

<b>res</b>	Reserved. Read as zero and ignored on write.
<b>WD</b>	Watchdog (WD) reset. When ERROR is asserted by the SSP, the MXCC sets the WD bit. Writing a 1 to this bit clears it. In the MBus configuration, CCERR will be asserted when WD is 1.
<b>SI</b>	Software Internal Reset. When this bit is written as 1, a software internal reset is generated. On software internal reset, the MXCC will issue a reset to the SSP and clear the WD bit. This bit is not cleared by the software internal reset and remains set after the software internal reset.

### 16.5.6 MXCC MBus Port Register

The MBus port register is a 32-bit register used in an MBus system to identify the processor number and to define the base address of the configuration space of each processor. It is read-only.

Figure 16–14. MXCC MBus Port Register



The fields of the MBus Port Register are:

<b>reserved</b>	Reserved. Read as zero and ignored on write.
<b>mid</b>	Module ID from the MID[3:0] pins.
<b>mdev</b>	MBus device number. This eight-bit field is hard-wired to 0x01.
<b>mrev</b>	Device revision number. This field contains a constant for each device revision. See the MXCC data sheet for revision numbers.
<b>mvend</b>	MBus Vendor number. This field contains the constant 0x4.

## 16.6 MXCC Block Copy Facility

The MXCC provides a facility to assist in the copying or zeroing of main memory. Transfers are performed using the system bus's block transfer operation. The transfers occur between memory and the stream data buffer. The stream unit is situated in the bus command logic, which is part of the cache controller core (see Figure 16-3).

Stream source and data addresses are physical addresses and may be cacheable or non-cacheable. Cacheable accesses snoop the local caches and perform cacheable bus accesses that should snoop other caches in the system.

To zero a block of memory, the stream data buffer is loaded with zeros. Then the destination address is stored into the stream destination register, which starts a block write of a sub-block from the stream data buffer to the memory address. A sub-block is 32 bytes in the MBus configuration and 64 bytes in the XBus configuration. If more than one sub-block of memory needs to be zeroed, additional destination addresses are stored into the stream destination register. All stream accesses are sub-block aligned.

To copy a block of memory, a sub-block address is written into the stream source register, and then a sub-block address is written into the stream destination register. This copies a sub-block from the source address to the destination address via the stream data buffer. Additional sub-blocks are copied in the same way. The stream hardware is interlocked to prevent a write to a stream register from completing while the register is still busy with a previously started operation. Therefore, a copy loop can write the stream source and destination register without checking whether the previous sub-block operation has completed. The block copy will proceed at the speed that sub-block can be transferred. When the stream copy source is not block-aligned, the lower bits are ignored, and the transfer completes on sub-block boundaries.

The block read facility may also prove useful in accomplishing a hardware-assisted software memory-scrubbing scheme.

The stream source and destination registers are interlocked only for accesses from the processor; writing into the source or destination register from the system bus before the previous operation is complete will have unpredictable results. Bus accesses to the stream data buffer should be attempted only while there is no processor-initiated stream operation in progress.

### 16.6.1 Stream Data Buffer

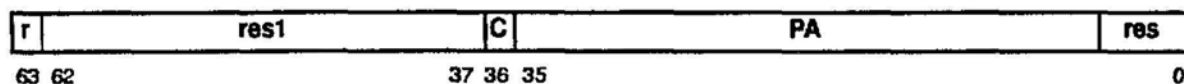
The stream data buffer is accessible through the MXCC's control space. This buffer is 64 bytes long and is read or written as double-words. The particular double-word of the buffer is selected by the DBL field of the control space address (see Figure 16-8). All stream accesses to the data buffer (see Subsections 16.6.2 and 16.6.3) transfer a sub-block of data. A sub-block is 64 bytes in XBus configurations and 32 bytes in MBus configurations. Only the first 32 bytes of the data buffer are used in MBus configurations.

### 16.6.2 Stream Source Register

The stream source register is a read/write register that controls stream read operations and reports their status. When an address is written into the register, it triggers a read of a sub-block from that physical address into the stream data buffer. A sub-block is 32 bytes on MBus and 64 bytes on XBus. The low-order address bits are ignored, and the transfers are always sub-block aligned.

The format of the stream source and destination registers is shown in Figure 16-15.

Figure 16-15. MXCC Stream Source / Destination Register



<b>r</b>	Ready. This bit indicates that the previous stream operation is completed. This bit is ignored when written.
<b>res1</b>	Reserved. Ignored on write and read as zero.
<b>C</b>	Cacheable. The C bit specifies whether the operation is to (or from) cacheable space. Cache-coherence checks are applied only to cacheable accesses. The access is to non-cacheable space when the C bit is clear.
<b>PA</b>	Physical Address. Physical address bits 35 to 6 for XBus or bits 35 to 5 for MBus of the source or destination. All transfers started from the Stream Source Register or Stream Destination Register are sub-block aligned in memory and in the stream data buffer and, except when errors are encountered, transfer an entire sub-block.

<b>res</b>	Reserved. Ignored for address generation. The bits written may be read. The size of this field varies with the bus configuration and is six bits wide in XBus configurations and five bits wide in MBus configurations.
------------	---

### 16.6.3 Stream Destination Register

The stream destination register is a read/write register that controls stream write operations and reports their status. When an address is written into the register, it triggers the write of a sub-block from the stream data buffer into that physical address. A sub-block is 32 bytes on MBus and 64 bytes on XBus. The low order address bits are ignored and transfers are always sub-block aligned.

The format of the stream destination registers, shown in Figure 16–15, is the same as that of the stream source register.

## 16.7 MXCC Interrupts

The MXCC supports interrupts differently on MBus than on XBus. In MBus configurations, system interrupts are sent to the MXCC on MIRL[3:0] pins and passed unmodified to the SSP on IRL[3:0]. The MXCC does not generate any interrupts.

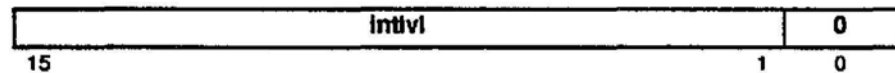
In an XBus system, interrupts from the system come via the XBus as interrupt request packets (see Section 19.7) or from the BootBus through hardware interrupt-polling cycles (see Section 20.4). The interrupt level indicated in the command is decoded, and the corresponding bit in the Interrupt Pending Register is set. The MXCC generates level-15 interrupts to report asynchronous errors. All the pending interrupts that are not masked by the interrupt mask register are prioritized, and the highest pending interrupt level is sent to the SSP on IRL[3:0].

In XBus configurations, the SSP can generate interrupts to be sent by XBus to the system. These interrupts are generated by writing to the interrupt generation register in the MXCC.

If the BootBus is not used in the XBus configuration, LDATA[3:0] should be pulled to ground to avoid generating unexpected interrupts.

### 16.7.1 Interrupt Pending Register (XBus)

Figure 16–16. XBus Interrupt Pending Register



**Intlvl** Interrupt Level. This field indicates pending interrupts on each of the 15 interrupt levels. Bit *n* is set if an interrupt is pending at level *n*. This field is read-only.

There is no level 0 interrupt, and thus never an interrupt pending at level 0. This register is read-only and is cleared through the interrupt pending clear register and by system reset. The values returned from reading the interrupt pending register are not predictable in the MBus configuration.

### 16.7.2 Interrupt Mask Register (XBus)

Figure 16–17. XBus Interrupt Mask Register



### INTMASK

Interrupt Mask. This field is used to selectively mask pending interrupts. If an interrupt is pending at level *n* and bit *n* of INTMASK is clear, the interrupt is prioritized and passed to the SSP. If bit *n* is set, pending interrupts at level *n* are not passed on to the processor.

This register is readable and writable. It is set to 1s by system reset (RSTIN).

The interrupt mask register has no effect on MBus configurations.

### 16.7.3 Interrupt Pending Clear Register (XBus)

Figure 16–18. XBus Interrupt Pending Clear Register



### INTCLR

Interrupt Clear. This field is used to selectively clear pending interrupts. Writing a 1 into bit *n* of this register clears the pending interrupt at level *n*.

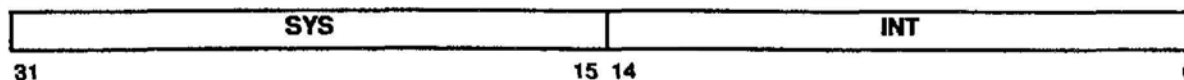
This register is write-only. A TO bus error is reported to the SSP for a read access to this register.

Writing the interrupt pending clear register has no effect in MBus configurations.

### 16.7.4 Interrupt Generation Register (XBus)

The interrupt generation register is a 32-bit write-only location used to generate system interrupts. It is accessible only from VBus. A write to the interrupt-generation register generates a two-cycle interrupt packet to the XBus. The first cycle contains the interrupt command code. The content of the interrupt generation register is in bits [31:0] of the second cycle. The entire 32 bits is passed to the BW for system-specific handling, but only the low-order 15 bits are recognized by the MXCC when received from the BW in an interrupt message. A TO bus error is reported to the SSP on a read access to this register.

Figure 16–19. MXCC XBus Interrupt Generation Register



The fields of the interrupt generation register are:

- |            |  |
|------------|--|
| <b>SYS</b> | System-Specific. Information passed to the bus watcher for system-specific use. It is not interpreted by either a sending or receiving MXCC or processor.  |
| <b>INT</b> | Interrupt Bits. Bits corresponding to the interrupt levels. Several bits may be set. If bit $n$ is set, an interrupt is to be communicated at level $n+1$ . Thus bit 0 corresponds to level 1, and bit 14 corresponds to level 15. |

Writing to the Interrupt Generation Register has no effect in MBus configurations.



## 16.8 MXCC Error Handling

The MXCC may encounter errors in operation. These can come from the bus, from parity checking on E-cache or bus data, from internal operations, or from illegal access.

Errors are handled in four different ways in MXCC:

- ☐ Errors logged to MXCC's error register and reported to SuperSPARC through encoding of control strobes **MEXC**, **RDDY** (or **WRDY**), and **RETRY** (see Table 18-1) are:
  - Errors on a read or a LDST operation,
  - Store exception pending condition of a write miss,
  - Data parity errors on VBus when the SSP is the master, and
  - Errors on a demap initiated by the SSP.
- ☐ Errors are reported to SuperSPARC through a level-15 interrupt (in XBus configurations only). Errors reported in this way are:
  - Asynchronous errors, which include errors of operations that have been acknowledged by MXCC to SuperSPARC. These include, for example, stream operations for Block Copy/Zero, shared writes in the XBus configuration, or non-cacheable writes in which errors occur later in the operation.
  - Data parity errors on the VBus when the MXCC accesses external cache for an incoming bus request.

All these errors are logged into the error register of MXCC. For the MBus configuration, these types of errors are reported to the system by asserting **AERR**.

- ☐ Errors are reported to the system by asserting **CCERR**. Errors reported in this way are:
  - XBus parity errors,
  - cache consistency errors, and
  - VBus parity errors on a flush operation.

These errors are considered catastrophic. They are logged into the error register of the MXCC before **CCERR** is asserted.

- ☐ Errors neither reported nor logged. For example, errors on the MXCC pre-fetch operation are ignored.

### 16.8.1 Bus Errors

Bus errors are signalled when an access cannot be satisfied for hardware reasons. This categorization excludes such MMU-detected conditions as protection violations.

The MXCC has four bus error codes. The MXCC preserves the four codes in signalling and logging the errors but does not differentiate the codes in any other way. Thus, meanings that a system actually assigns to the codes may differ from those suggested here.

The codes are:

<b>TO</b>	Timeout. This code is returned when the addressed location does not return any answer after some fixed amount of time.
<b>BE</b>	Bus Error. This code is returned when the addressed location rejects the required action because it is illegal.
<b>UC</b>	Uncorrectable Error. This code is returned when the addressed location rejects the required action because of an internal failure.
<b>UD</b>	Undefined (other) Error. This code is returned for errors that cannot be classified under the other codes.

### ***Bus Errors on Instruction Fetches***

An instruction fetch accesses an instruction that will be executed. If a bus error is encountered on an instruction fetch, an `instruction_access_exception` trap is taken by the SSP when the instruction reaches the execution stage. Errors on the system bus are passed through MXCC to the SSP and logged in CCER.

The SSP's Fault Status Register (see Subsection 9.12.3) contains the bus error code in the UD, UC, TO, and BE bits, with FT=5, AT=2 or 3, and FAV=0. The virtual address that encountered the error is the trap PC (see Chapter 12). The fault address register (see Subsection 9.12.4) is not updated.

### ***Bus Errors on Instruction Prefetches***

An instruction prefetch accesses an instruction that may or may not be executed in the future. If a bus error is encountered on an instruction prefetch, the prefetch operation is aborted, and the error is ignored.

### ***Bus Errors on Data Loads***

A data load is the access made by a load instruction. If a bus error is encountered on a data load, a `data_access_exception` trap is taken by the SSP. Errors on the system bus are passed to the SSP by the MXCC and logged into the CCER.

The SSP's Fault Status Register (MFSR) (see Subsection 9.12.3) contains the bus error code in the UD, UC, TO, and BE bits with FT=5, AT=0-3, and FAV=0. If the erroring access is through an ASI other than 0x8-0xB and 0x20-0x2F, MFSR.CS is set to 1. The virtual address that encountered the error is saved in the SSP's MFSR (see Subsection 9.12.4).

### **Bus Errors on Synchronous Data Stores**

A synchronous data store is an access made by a store instruction that cannot be buffered in the SSP's store buffer. The LDSTUB, LDSTUBA, SWAP, and SWAPA instructions always perform synchronous stores. In addition, STA (or STDA) instructions to ASIs other than 0x08-0x0B and 0x20-0x2F are also synchronous. See Section 10.6 for more information on synchronous stores. The MXCC passes errors on the system bus to the SSP and logs the error on CCER.

The SSP's MFSR (see Subsection 9.12.3) contains the bus error code in the UD, UC, TO, and BE bits, with FT=5, AT=4-7, and FAV=1. If the erroring access is through an ASI other than 0x8-0xB and 0x20-0x2F, MFSR.CS is also set. The virtual address that encountered the error is saved in the SSP's fault address register (MFAR) (see Subsection 9.12.4).

### **Bus Errors on Asynchronous Data Stores**

An asynchronous data store is any store buffered in the SSP's store buffer. (See Chapter 8 and Section 10.6.) The handling of bus errors on asynchronous data stores depends on the state of the SSP's store buffer and on whether the error is early or late.

An early error is an error that is signalled to the SSP before the bus operation is effectively started by the MXCC. There are two cases of early store errors:

- ☐ Memory store that misses in the external cache and for which the cache receives a bus error indication from memory when it tries to load the missing cache block.
- ☐ VBus parity error.

If the SSP's store buffer is disabled and an early error is notified, the error is handled as a synchronous data store error.

If the SSP's store buffer is enabled and an early error is notified, the processor takes a data\_store\_error trap. The MFSR.SB bit is set but the bus error code is not logged in MFSR, and MFAR is not updated.

A late error is an error that is signalled to the SSP after the bus operation has been started by the MXCC and acknowledged to the SSP. Late errors occur only on non-cacheable stores.

If a late error is signalled in an XBus configuration, a level-15 interrupt is signalled by the MXCC to its directly connected SSP. The CCER contains the error information with the AE bit set to 1, and the DCmd subfield of CCER.CCOP is set from the operation. In MBus configurations, the AERR signal is asserted instead of a level-15 interrupt being signalled.

The value of CCCR.MC (multiple command enable) does not affect error handling of store operations.

### **Bus Errors on Block Copy Operations**

Block copy operations are initiated by using the MXCC's stream registers (see Section 16.6). If a bus error is signalled during a block copy operation, a level-15 interrupt is signalled by the MXCC to its directly connected SSP. The MXCC error register (CCER) is updated with the error information, and CCER.AE is set to 1 to indicate an asynchronous error.

Block copy errors may be differentiated from late asynchronous errors by the value logged in the DCmd subfield of CCER.CCOP. Table 16-14 shows CCER.CCOP.DCmd values, the operation in progress, and the XBus packet name that indicates the error if using XBus. The DCmd field is set to the same values when in an MBus configuration.

*Table 16-14. DCmd Field of Block Copy and Asynchronous Errors*

DCmd	Operation	Packet
0x05	Block Load	Stream Read Reply
0x15	Block Load	IO Stream Read Reply
0x0F	Block Store	Stream Write Reply
0x17	Block Store	IO Stream Write Reply
0x13	Asynchronous Store Error	IO Write Reply

### **Bus Errors on MMU TLB Operations**

An MMU TLB operation is a memory reference initiated by the SSP's memory management unit to load entries into its TLB or to set the modified or referenced bits of a PTE (see Chapter 9). If a bus error is signalled during an MMU TLB operation, the SSP takes either an `instruction_access_exception` trap or a `data_access_exception` trap, depending on whether the operation was initiated using ASIs 0x08-0x09 (instruction fetch ASIs) or another ASI. Errors on the system bus are passed through the MXCC to the SSP and are logged in the CCER.

If an `instruction_access_exception` trap is taken, the SSP's MFSR is updated with the bus error code in the UD, UC, TO and BE bits, FT=4, and FAV=0. The virtual address of the error is the trap PC, which is saved as normal for the trap. The MFAR is not updated.

If a `data_access_exception` trap is taken, the SSP's MFSR is updated with the bus error code in the UD, UC, TO, and BE bits, FT=4 and FAV=1. If the access was through an ASI other than 0x08-0x0B or 0x20-0x2F, MFSR.CS is set to 1. The virtual address of erroring operation is saved in the MFAR.

### Illegal Access

Illegal access is an access to an internal MXCC register or memory with an invalid address or operation. Examples of illegal accesses from VBus include:

- ☐ Atomic load-store to BootBus or the MXCC registers.
- ☐ Out-of-range control space access.
- ☐ Read of interrupt generation register.

Invalid access from either the SSP or VBus can be reported as:

- ☐ TO error to SuperSPARC.
- ☐ A TO error if the system bus has timeout logic. (Illegal accesses from the system bus side are ignored.)

### Bus Errors on Outgoing Transactions

In the MBus configuration, an error on an outgoing request is reported back to the MXCC with the MBus acknowledgment type by encoding the type of the error into `MRDY`, `MERR`, and `MRTY`, as shown in Table 17-3.

In the XBus configuration, an error on an outgoing request is reported to the MXCC in two different ways (see Subsection 19.5.6):

- ☐ The error bit in the header cycle of the reply packet is set. The second cycle carries the error code in the three least significant bits.
- ☐ Odd parity is used on a data cycle to indicate a memory fault data cycle. In this case, the three least significant bits of the memory fault data cycle contain the error code.

These errors are signalled to the local SSP (which initiated this request), as described above according to the type of access.

### ***Parity Errors on Incoming Requests***

In MBus and XBus configurations, if a parity error occurs on the VBus when the MXCC accesses external cache in response to an incoming bus request, a VBus parity error will be reported to the requestor as an uncorrectable error.

### ***Parity Error on VBus when the Processor is Master***

A parity error on VBus when SuperSPARC is the bus master is reported to SuperSPARC as an undefined error.

### **16.8.2 MXCC Level-15 Interrupt and CCERR/AERR**

In XBus configurations, the MXCC signals a level-15 interrupt to the locally connected processor when an asynchronous error is detected. The asynchronous error may have been on an operation initiated by the local SSP (for example, a late error in an asynchronous data store), or on an operation initiated from another processor or an I/O device, but which was detected by the MXCC (for example, E-cache parity error on data accessed because the E-cache is the owner).

When a local level-15 interrupt is issued, the error information is logged in the CCER.

In MBus configurations, these same errors cause AERR signal to be asserted to signal the error to the system. AERR in MBus configurations is the same pin as CCERR in XBus configurations.

Some errors are always reported to the system by asserting CCERR or AERR. Errors reported in this way are: XBus parity errors, cache-consistency errors, and VBus parity errors on a flush operation. These errors are considered catastrophic. They are also logged into the error register of the MXCC.

### **16.8.3 Errors Detected by MXCC**

The MXCC detects parity errors on both XBus and VBus. It can also detect cache-consistency errors in XBus configurations.

### ***External Cache Parity Error***

A parity error in the external cache can be discovered in three cases, each of which is handled differently.

- ☐ If another processor reads data from the E-cache because the E-cache is the owner, a parity error is signalled to the bus as an uncorrectable (UC) error. In XBus configurations, the locally attached processor is signalled with a level-15 interrupt. In MBus configurations, the error is reported by asserting the AERR signal. The error information is logged in CCER; CP is set to 1.

- ☐ If the MXCC detects an E-cache parity error when trying to write back a block to main memory due to the replacement algorithm, it proceeds with the write of the incorrect data to memory. In XBus configurations, it signals the locally attached processor with a level-15 interrupt. In MBus configurations,  $\overline{\text{AERR}}$  is asserted. The error information is logged in the CCER; CP is set to 1.
- ☐ If an SSP detects an E-cache parity error during a read from its own E-cache, the error condition is handled as if a bus error indication UC had been signalled on the read operation. The SSP's MFSR.P bit is set to 1.

The first two cases can be distinguished by the value logged into the DCmd subfield of CCER.CCOP. The DCmd subfield is 0x06 for the second case (E-cache parity error on write back) only.

### **SSP Write Parity Error**

If an E-cache (VBus) parity error is detected by MXCC during a write by the SSP to the E-cache, an early bus error indication of an undefined (UD) error is signalled to the processor. The VP bit in the CCER is set to 1.

### **XBus Parity Error**

When the MXCC detects an XBus parity error, it sets the XP bit of the CCER and asserts the  $\overline{\text{CCERR}}$  signal to the system. See Chapter 19.

### **Cache Consistency Errors**

Cache-consistency errors are detected by using the encodings of the XBus packet's Tag Command (TCMD) field that indicated expected tag value. A mismatch in tag value from the expected value triggers a cache-consistency error. When the MXCC detects a cache-consistency error, it sets the CC bit in the CCER and asserts the  $\overline{\text{CCERR}}$  signal to the system. Proper operation of MXCC after a cache-consistency error is not assured; system logic should reset the MXCC after this error occurs.

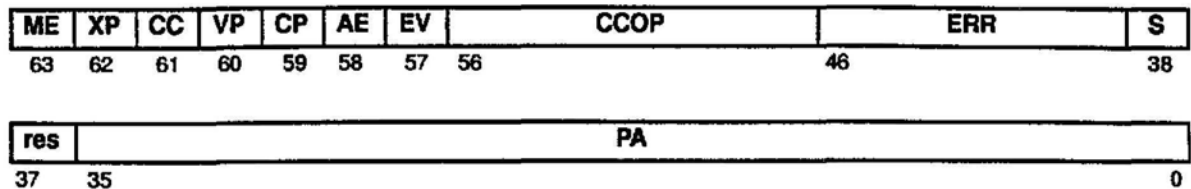
Cache-consistency errors cannot occur in MBus configurations.

#### **16.8.4 MXCC Error Register (CCER)**

The MXCC error register is a 64-bit register that records the address and other relevant information from a transaction that generates an error. The error register is readable, writable, and JTAG-scannable. It is not cleared at system reset (RSTIN).

The format of the MXCC Error Register is shown in Figure 16–20. Bits in this register are cleared by writing 1s to the relevant field. This allows the system to clear previously recognized errors by storing the previously read contents back into the register. The CCOP, ERR, S, and PA fields are latched on the first occurrence of a CC, CP, or AE error. This information will not be updated on subsequent errors until the EV bit is cleared.

Figure 16–20. MXCC Error Register (CCER)



The fields of the error register are:

- |           |   |
|-----------|---|
| <b>ME</b> | Multiple Errors. This bit indicates that multiple errors of the same type occurred. ME is set when an error occurs that would set XP, CC, VP, CP, or AE with that bit already set.  |
| <b>XP</b> | XBus Parity Error. When a parity error is detected on the XBus, this bit is set to 1 if it is not previously set. If previously set, ME is set instead. CCERR is also asserted.   |
| <b>CC</b> | Cache-Consistency Error. When the MXCC detects an unexpected E-cache tag status, this bit is set to 1 if not previously set. If previously set, the ME bit is set to 1 instead. If the EV bit is not already set when the CC error occurs, it is set now, and the parity bits are stored in ERR. The CCOP, S, and PA fields are set to reflect the operation code, the supervisor state, and the physical address of the command that caused the error. CCERR is also asserted. |
| <b>VP</b> | VBus Parity Error when the SSP is the bus master. When parity errors are detected on the VBus during a processor-initiated write operation, this bit is set to 1 if not previously set. If previously set, the ME bit is set to 1 instead. The MXCC reports the error to SuperSPARC with an undefined error (MEXC = H, WRDY = L, RETRY = L) acknowledgment on VBus (see Chapter 18).  |



<b>CP</b>	<p>VBus Parity Error with the MXCC as the bus master. When parity errors are detected on VBus for an E-cache access initiated from the system bus, this bit is set to 1 if not previously set. If previously set, the ME bit is set instead. If the EV bit is not already set when the CP error occurs, it is set now, and the parity bits are stored in ERR. The CCOP, S, and PA fields are set to reflect the operation code, the supervisor state, and the physical address of the command that caused the error. An error reply with the error code set to uncorrectable error is then sent to the requester. In XBus configurations, a level-15 interrupt is sent to the SSP. In MBus configurations, CCERR is asserted.</p>
<b>AE</b>	<p>Asynchronous Error. When an error occurs on a write or stream operation for which the MXCC has previously sent an acknowledgement to the SSP, this bit is set to 1 if not previously set. If previously set to 1, the ME bit is set instead. If the EV bit has not already been set when the AE error occurs, it is set now. The CCOP, S, and PA fields are set to reflect the error code. In XBus configurations, a level-15 interrupt is sent to the SSP. In MBus configurations, CCERR is asserted.</p>
<b>EV</b>	<p>Error Information Valid. This bit is set when the CCOP, ERR, S, and PA are loaded with information about an error. These fields are updated with information from a new error condition only when EV is clear. Error handling should clear EX after reading the error information.</p>
<b>CCOP</b>	<p>MXCC Operation Code. The MXCC operation code of the command that incurred the error. In the XBus configuration, CCOP contains bits from the header cycle of a reply command. See Subsection 16.8.5.</p>

**ERR**

Error code. This field is used to log the error code of an XBus reply packet or MBus acknowledgement, or the parity bits on the VBus when the MXCC is the master for CP errors. The error codes from bus errors appear in ERR[2:0] and are encoded as shown in Table 16-15. Parity bits are logged in ERR, with a one-to-one correspondence with the DPAR[0:7] signals where ERR[7] corresponds to the DPAR[0] pin.

Table 16-15. CCER.ERR Error Codes

Error Code	Meaning
0	reserved
1	UC: Uncorrectable Error
2	TO: Time out
3	BE: Bus Error
4	UD: Undefined error
5-7	reserved

**S**

Supervisor bit. This bit records the state of the SU signal on VBus for commands initiated by the SSP or the SU bit in MBus command words from other processors. It is set on CP and AE errors. This is used only in the MBus configuration. In XBus configurations, it should be ignored when reading the error register.

**res**

Reserved. Read as zero and ignored on writes.

**PA**

Physical Address of the access causing the error.

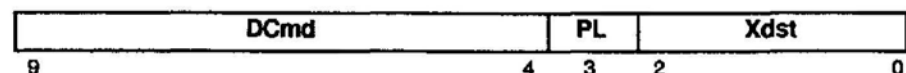
**Note:**

Errors occurring on the MBus during writebacks cause the last writeback address to be latched, not necessarily the address that got the error.

### 16.8.5 CCOP Sub-fields

The CCOP field of CCER has additional substructure. The subfields of CCOP are shown in Figure 16-21.

Figure 16-21. Format of CCOP Field



<b>DCmd</b>	Data Command. This command is derived from the packet header. The low-order bit of this field indicates that the command was a reply. If this bit is clear, the command was a request. The encoding of this field is given in Subsection 19.7.1.
<b>PL</b>	Packet length (0 is two-cycle, and 1 is nine-cycle).
<b>Xdst</b>	Destination ID. See Chapter 19, XBus, for more details.

### 16.8.6 Interpretation of CCER After Errors

After an error, CCER logs the cause of the error. If CCER.ME is set, the CCER contains information on more than one error. If there is a single error logged, interpretation is simplified. The main decoding of the error is performed by examining the XP, CC, VP, CP, and AE bits. If ME is not set, only one of these bits should be set. The paragraphs below provide information on decoding the information in CCER in each of these cases.

It is important to read the CCER as soon as practical and clear the error indicators by storing the value read back into the CCER. The errors can be processed from the record in the value already read from CCER. Errors occurring later will be logged into CCER without interfering with the interpretation of the previously read error information.

#### Asynchronous Errors (AE)

When the AE bit is set, an asynchronous error has occurred. If neither the CC nor the CP bit is also set, ME is clear and EV is set. The CCOP, S<sub>c</sub>, and PA fields contain information on the operation that encountered the error.

Table 16–16 indicates some of the most common of the possible values in the CCER.CCOP.DCmd field after an asynchronous error is logged (AE=1).

Table 16–16. DCmd Field After Asynchronous Error

DCmd	Command	Operation Causing Error
0x05	NC Get Block Reply	Block load from memory space
0x0F	Put Block Reply	Block store to memory space
0x13	IO Put Single Reply	Store to I/O space
0x15	IO Get Block Reply	Block load from I/O space
0x17	IO Put Block Reply	Block store to I/O space

### Cache Parity Errors (CP)

When the CP bit is set, an asynchronous error has occurred. If neither the CC nor the AE bit is also set, ME is clear, and EV is set, then the CCOP, S, and PA fields contain information on the operation that encountered the error.

Table 16-17 indicates some of the most common of the possible values in the CCER.CCOP.DCmd field after a cache parity error is logged (CP=1).

Table 16-17. DCmd Field After Asynchronous Error

DCmd	Command	Operation Causing Error
0x04	NC Get Block Rqst	XBus NCGetBlk (e.g., bcopy from another MXCC)
0x06	Flush Block Rqst	Write back by E-cache to memory
0x0C	Get Block Rqst	XBus GetBlock (e.g., access by another MXCC to an owned block)
0x10	IO Get Single Rqst	Foreign read to cache data via XBus operation

### Processor Parity Errors (VP)

If the MXCC detects a VBus parity error when the SSP is writing to E-cache, the MXCC sets VP and gives an early bus error indication of UD to the processor. Other error information is not logged in CCER. The processor will take a data\_access\_exception on the access and may retry the store.

### XBus Parity Errors (XP)

When the MXCC detects an XBus parity error, it sets the XP bit of the CCER. Other error information is not recorded in CCER. For full recovery or logging, external logic should log the failing bus transaction.

### Cache Consistency Errors (CC)

When the CC bit is set, an asynchronous error has occurred. If neither the CP nor AE bit is also set, ME is clear, and EV is set, then the CCOP, S, and PA fields contain information on the operation that encountered the error.

The TCM that caused this error is not preserved in CCER.

The MXCC should be reset after a Cache-Consistency Error is detected, since proper operation is not assured after a cache-consistency error.



# **MBus**

The SPARC MBus is a high-speed interface that connects SPARC processor modules to physical memory and I/O modules. The MBus is not intended for use as a general expansion bus on a system back-plane spanning numerous boards. Rather it is intended to operate in a carefully controlled geographical area with the interconnect and associated circuitry located on only one printed wiring board (PWB). Processors may be located on the main PWB or on a small module. Modules consist of one or more integrated circuits, on one (or more) of which the MBus interface is contained.

This chapter covers MBus operation in two SuperSPARC configurations: connecting a SuperSPARC processor (SSP) directly to the MBus and using a module containing an SSP and a MultiCache Controller (MXCC) chip. In each configuration we consider how the processor or module behaves as a bus master, bus slave, and bus snoop.

<b>Topic</b>	<b>Page</b>
<b>17.1 MBus Overview .....</b>	<b>17-2</b>
<b>17.2 SuperSPARC MBus Operation .....</b>	<b>17-4</b>
<b>17.3 MBus Signals .....</b>	<b>17-6</b>
<b>17.4 MBus Operation .....</b>	<b>17-13</b>
<b>17.5 SuperSPARC MBus Transactions (Generic) .....</b>	<b>17-19</b>
<b>17.6 MBus for SuperSPARC Without the MXCC .....</b>	<b>17-33</b>
<b>17.7 SuperSPARC With the Multicache Controller on MBus .....</b>	<b>17-41</b>
<b>17.8 MBus Timing Summary .....</b>	<b>17-48</b>

## 17.1 MBus Overview

MBus is a SPARC International standard designed to offer processor-independent connections between one or more processors and memory. MBus standard connectors on a system board can accept MBus processor modules, offering performance upgradable systems. The *MBus Specification* is the specification of the MBus logical and electrical protocols and is available from SPARC International.

MBus is a high-performance bus. It is fully synchronous; all transfers are controlled by an MBus clock, which can be up to 40 MHz. It supports block transfers in sizes up to 128 bytes with a peak transfer rate of 320 MBps.

MBus is a 64-bit bus. The 64 data lines are multiplexed, carrying a command word to start each transaction and data to complete it. The MBus command word contains the memory address of the transaction, the transaction type (e.g., read or write), its size (e.g., 1 byte or 32 bytes), and other information.

The unit that initiates a transaction on the bus is termed the "master." The unit that is addressed in a transaction is termed the "slave."

Each MBus system requires an MBus arbiter. Each unit that can be a master has a request signal to the arbiter and a grant signal from the arbiter. Units can use the bus when granted if the bus is free. MBus does not define the arbitration algorithm to be used; it is system-dependent.

MBus is defined for uniprocessor and multiprocessor systems. The uniprocessor form of MBus is termed "Level 1," and the multiprocessor "Level 2." SuperSPARC configurations (both with and without the MXCC) use the Level 2 signals and protocols (except when all caches are disabled). This chapter will focus on Level 2. All discussion of MBus is assumed to be MBus Level 2 unless explicitly marked as MBus Level 1.

MBus Level 2 supports coherent cache multiprocessing. When the system consists of only caches and memory controllers designed for the MBus Level 2 protocols, all caches are kept coherent on the bus. There are several compatibility requirements for coherent caches that must interoperate on MBus. See Subsection 17.4.4.

The cache coherence protocol used in MBus is similar to the MOESI protocol used in Futurebus. This type of protocol differentiates between data that is shared and data that is held exclusively. It also differentiates between data that is held unmodified and data that has been modified.

Two special signals on MBus Level 2 are used in the operation of the cache-coherence protocol, **MSH** (shared) and **MIH** (inhibit). **MSH** is asserted by any cache that has the addressed data. **MIH**, when asserted on a read, inhibits the memory from returning data. A cache that has the data will supply it instead.

Coherent transactions on MBus always have the transaction sizes set to 32 bytes, the specified size of a cache sub-block.



## 17.2 SuperSPARC MBus Configurations

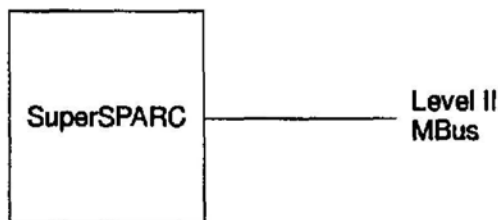
Table 17-1 exemplifies several SuperSPARC design configurations.

Table 17-1. SuperSPARC Chipset MBus Configurations

Configuration	Super- SPARC	SRAMs	MXCC	Bus	MP Ready
Direct MBus	√			MBus	√
MXCC no SRAM MBus	√		√	MBus	√
Full Module MBus	√	√	√	MBus	√

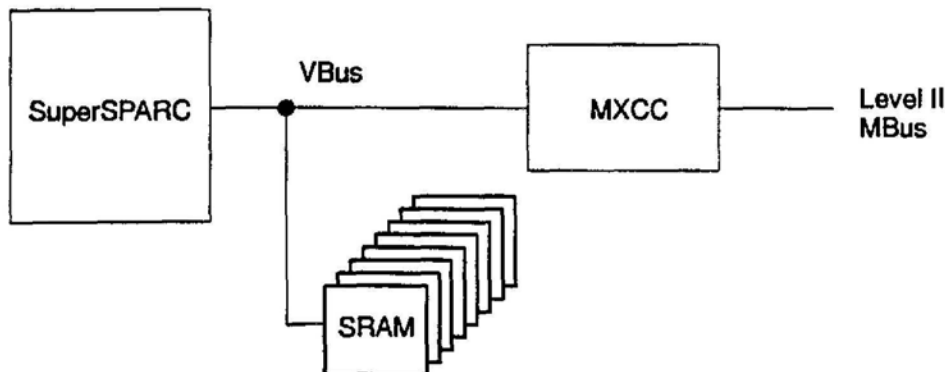
The direct MBus configuration uses the SuperSPARC microprocessor without the MXCC or external SRAMs. In this configuration the SuperSPARC is connected directly to MBus, as shown in Figure 17-1, and must run at the MBus clock frequency.

Figure 17-1. Direct MBus MBus System



The Full Module MBus system is diagrammed in Figure 17-2. The external cache memory provides significant performance improvement and greatly decreases bus traffic in order to support more processors on a system bus.

Figure 17-2. Full-Module MBus System



In a non-MBus system, SuperSPARC supports external system bus interfaces in its XBus configurations. See Chapter 19.

External cache (E-cache) RAM may not be needed in every application. This configuration is listed as the "MXCC no SRAM" configuration in Table 17-1.

The E-cache is organized as a direct-mapped copy-back cache with a size of 1 Mbyte. This configuration is implemented with eight 128Kx8 or 128Kx9 synchronous SRAMs. The 128K x 9 SRAMs are needed to implement byte parity on the E-cache data storage. Parity is directly supported by both SuperSPARC and the MXCC.

Synchronous SRAMs have registers on each input and output. This allows pipelined operation. An address is presented to an SRAM before the active clock edge, and it is registered in the SRAM at the clock edge. The SRAM reads out the addressed location before the next active clock edge, and the result is stored in an output register at the next edge. New addresses can be supplied at each clock edge, and new outputs appear after two clock periods of delay. Writing works similarly with address, data, output enable, and write-enable being registered on the active clock edge and stored in the internal array during the subsequent clock period.

The synchronous SRAMs used in the full module configurations are supplied by several manufacturers.

## 17.3 MBus Signals

### 17.3.1 Physical Signal Summary

Table 17-2 summarizes all the MBus physical signals. A bar over the signal name indicates the signal is active low (negative logic). A signal type of BS signifies bi-state, TS signifies three-state, and OD signifies open-drain.

Table 17-2. MBus Physical Signal Summary

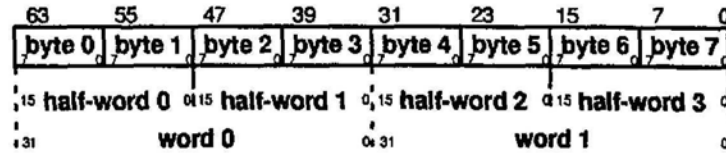
Signal	Description	Line Type	Signal Type
MCLK	MBus Clock	dedicated	BS
MAD[63:0]	Address/Control/Data	bussed	TS
MAS	Address Strobe	bussed	TS
MRDY	Data Ready Indicator	bussed	TS
MRTY	Xaction retry Indicator	bussed	TS
MERR	Error Indicator	bussed	TS
MSH	Shared (level-2)	bussed	OD
MIH	Inhibit (level-2)	bussed	TS
MBR	Bus Request	dedicated	BS
MBG	Bus Grant	dedicated	BS
MBB	Bus Busy Indicator	bussed	TS
IRL[3:0]	Interrupt Level	dedicated	BS
ID[3:0]	Module Identifier	dedicated	BS
AERR	Async. Error out	bussed	OD
RSTIN	Module Reset In	impl. dep.	BS

The MBus fieldnames are as follows:

**MCLK** MBus Master Clock.

**MAD[63:0]** Memory Address and Data. During address phase, MAD[35:0] will carry the physical address, while MAD[63:36] will carry the information specific to the transaction requested, as described in Subsection 17.3.2. During the data phase, MAD[63:0] contains data as shown below. Data must be aligned for transactions involving fewer than eight bytes (a double-word). For example, a word that has an even word address must be sent on MAD[63:32], whereas an odd-addressed word must be sent on MAD[31:0]. This is termed "big-endian" data order.

Figure 17-3. MBus Byte Alignment



<b>MAS</b>	Memory Address Strobe. This signal is asserted by the bus master during the first cycle of a bus transaction. This cycle is referred to as the "address cycle." All other timing is relative to MAS. For example, A+3 refers to the third cycle after MAS is asserted.
<b>MRDY</b>	MBus Ready. This signal is used as one of the three bits encoding the transaction status, as shown in Table 17-3. If only MRDY is asserted, valid data has been transferred. The three status bits (MRDY, MRTY, and MERR) are normally asserted by the addressed slave.
<b>MRTY</b>	MBus Retry. This signal is used as one of the three bits encoding the transaction status, as shown in Table 17-3. If only MRTY is asserted, the slave wants the master to abort the current transaction immediately and start over. The master will relinquish the bus after this retry acknowledgement.
<b>MERR</b>	MBus Error. This signal provides one of the three bits that encode the transaction status, as shown in Table 17-3. The encoding with only MERR asserted indicates that a bus error has occurred.

**Note:**

If any type of acknowledgement other than Valid Data Transfer is issued, the cycle in which it is issued is the last cycle of the transaction, regardless of how many more acknowledgement cycles would normally occur.

Table 17-3. Transaction Status Bit Encoding

MERR	MRDY	MRTY	Meaning
H	H	H	Idle cycle
H	H	L	Relinquish and Retry
H	L	H	Valid Data Transfer
H	L	L	reserved
L	H	H	ERROR1 → Bus Error
L	H	L	ERROR2 → Timeout
L	L	H	ERROR3 → Uncorrectable
L	L	L	Retry

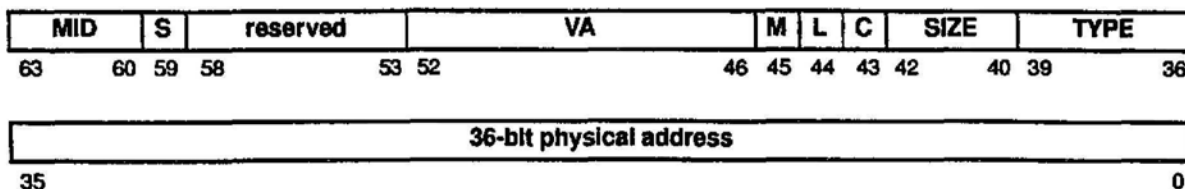
<b>MBR</b>	MBus Request. This signal is asserted by an MBus master to request bus ownership. There is a unique MBR from each master to the MBus arbiter.
<b>MBG</b>	MBus Grant. This signal is asserted by the external MBus arbiter when the particular master is granted the bus. There is a unique MBG signal per master.
<b>MBB</b>	MBus Busy. This signal is asserted as an output during the entire transaction, from the initial assertion of MAS to the assertion of the last MRDY or other termination acknowledgement (such as an error acknowledgement).
<b>MIF</b>	Memory Inhibit. This is a level-2 MBus signal. MIF is asserted by the owner of a cache block after it receives its address. This informs main memory that the current CR or CRI request should be ignored. See Section 17.8 for the system-dependent last MIF cycle.
<b>MSH</b>	Cache Block Shared. This is a level-2 MBus signal. Whenever a CR transaction appears, each module on the MBus should search its cache directory. If a valid copy is found, MSH should be asserted. MSH timing is system-dependent; see Section 17.8.

<b>RSTIN</b>	Module Reset Input. This signal should reset all logic on a MBus module to its initial state. At power-on, RSTIN must be held low for at least 100ms for both SuperSPARC and the MXCC. See Chapter 13 for more specific details.
<b>AERR</b>	Module Asynchronous Error Detect. This signal is asserted by the module to indicate that an internal error was detected. SuperSPARC will enter error mode when any exception is taken with traps disabled (PSR.ET = 0). Once in error mode, SuperSPARC will automatically perform a watchdog reset. AERR will remain asserted until the MFSR.EM bit is read and thus cleared. See Section 12.5. AERR is also asserted on a store buffer error. AERR remains asserted until MFSR.SB is cleared. See Subsection 10.6.5.
<b>IRL[3:0]</b>	Interrupt Request Level. These pins carry the interrupt request level to the processor. Each SuperSPARC module receives a dedicated set of IRL[3:0]. The processing of these signals is described in Section 12.7.
<b>ID[3:0]</b>	Module Identifier. These pins carry the module identifier. This identifier is asserted during the address phase of every transaction on MID[63:60]. The identifier is also used to identify a unique address range for module identification, initialization, and configuration.

### 17.3.2 MBus Command Word Signals

The MAD[63:00] signals contain multiplexed information. During the address phase of the MBus transaction, when  $\overline{MAS}$  is asserted, the command word information is presented by the master to the slave. During the data phase of the MBus transaction, the MAD[63:00] signals are used to transmit data. Figure 17-4 shows the format of the command word. The following section describes these signals.

Figure 17-4. MBus Command Word Format



<b>MID</b>	MBus Module Identifier—MAD[63:60]. This field is the ID[3:0] of the master for this transaction. The MID is sourced by all MBus modules and allows slaves to selectively reconnect after issuing a Relinquish and Retry.
<b>S</b>	Supervisor Access Indicator—MAD[59]. This signal is asserted to indicate that the MBus transaction was initiated by a processor in supervisor state. This is an advisory bit, not used by MBus transactions, possibly of use to the slave device.
<b>reserved</b>	Reserved. MAD[58:54] are driven high and not used at this time.
<b>VA</b>	Virtual Address bits 19 through 12—MAD[53:46]. This field is used by some processors with virtually indexed caches. This field is not used by the SSP or the MXCC, as they use physically addressed caches. These bits are driven high when the transaction originates from an SSP or MXCC and are ignored by the SSP or MXCC when snooping or when addressed as a slave.
<b>M</b>	Boot Mode/Local Bus Indicator—MAD[45]. This signal is asserted by the processor module during the address phase of boot mode transactions. The signal is also asserted during local bus transactions. This is an advisory bit and is not used by MBus transactions. When SuperSPARC is connected directly to MBus, this bit is used to indicate boot mode. When SuperSPARC is used with an the MXCC, the MXCC cannot know that SuperSPARC is in boot mode; the MXCC therefore never asserts this bit.

<b>L</b>	Lock Indicator—MAD[44]. This bit is an advisory bit to indicate resource locking. It is intended to lock a slave to a particular master. The bus itself is locked by maintaining MBB asserted during the entire operation. SuperSPARC asserts the lock bit when performing atomic operations (SWAP or LDSTUB), or when performing a page table walk. This bit is set by the MXCC when performing a non-cached LDST transaction initiated from the VBus. It is also set when the MXCC performs a CR or a CRI operation that has write-back operations pending or during any write back operation.
<b>C</b>	Cacheable indicator—MAD[43]. When This signal is asserted, it indicates data is cacheable.
<b>SIZE</b>	SIZE[2:0]—MAD[42:40]. Indicates the size of operation in bytes. (See Table 17-4.) Neither MXCC nor SSP ever initiates transactions with SIZE > 32 bytes.

Table 17-4. SIZE Format

SIZE[2]	SIZE[1]	SIZE[0]	TRANSACTION SIZE
L	L	L	Byte
L	L	H	1/2 Word (2 Bytes)
L	H	L	Word (4 Bytes)
L	H	H	Double Word (8 Bytes)
H	L	L	16 Byte Burst
H	L	H	32 Byte Burst
H	H	L	64 Byte Burst†
H	H	H	128 Byte Burst†

†not supported in SSP or MXCC

<b>TYPE</b>	TYPE[3:0] – MAD[39:36]. Indicates the type of operation as illustrated in Table 17-5.
-------------	---



*Table 17-5. TYPE Format*

TYPE[3]	TYPE[2]	TYPE[1]	TYPE[0]	SIZE	TRANSACTION TYPE
L	L	L	L	any	Write
L	L	L	H	any	Read
L	L	H	L	32B	Coherent Invalidate (CI)
L	L	H	H	32B	Coherent Read (CR)
L	H	L	L	32B	Coherent Write and Invalidate (CWI)
L	H	L	H	32B	Coherent Read and Invalidate (CRI)
L	H	H	L	-	reserved
L	H	H	H	-	reserved
H	X	X	X	-	reserved

**physical address**

This represents the 36-bit physical address on MAD[35:0].

## 17.4 MBus Operation

The MBus specification has two levels of compliance: Level 1 and Level 2. Level 1 includes basic MBus signals and transactions needed to design a complete uniprocessor system. Level 2 introduces additional signals (two) and transactions needed to design a cache-coherent, shared-memory multiprocessor.

### Level 1

The level 1 MBus supports two transactions: read and write. These transactions simply read or write a specified size of data from a specified physical address. These transactions are supported using a subset of MBus signals—namely, a 64-bit multiplexed address/data bus (MAD[63:0]), an address strobe signal (MAS), and an encoded acknowledge on three signals (MRDY, MRTY, and MERR). Additional level 1 signals support arbitration for modules (MBR, MBG, and MBB), as well as interrupts (IRL[3:0]), reset (RSTIN and RSTOUT), asynchronous errors (AERR), and module identification (ID[3:0]). The MBus reference clock (CLK) completes the signal requirements for a Level 1 system.

MBus assumes that there are central functional elements to perform reset, arbitration, interrupt distribution, time-out, and MBus clock generation.

### Level 2

The level-2 MBus includes all Level 1 transactions and signals and adds four transactions and two signals to support cache coherency. This is to facilitate the design of shared-memory multi-processor systems. In Level 1, details of the caches inside modules are not visible to the MBus Transactions. This changes with Level 2, where many of the aspects of the caches are assumed as part of the new MBus transactions. To participate in cache-consistent sharing using Level 2 transactions, a cache must have a “write-back” policy, an “allocate” policy on write misses, and a block or sub-block size of 32 bytes. Cache lines are assumed to have at least five states (invalid, exclusive clean, exclusive dirty, shared clean, and shared dirty).

The additional transactions present in Level 2 systems are Coherent Read (CR), Coherent Invalidate (CI), Coherent Read and Invalidate (CRI), and Coherent Write and Invalidate (CWI). The two additional signals are shared (MSH) and inhibit (MIH). All coherent transactions have SIZE of 32 bytes, except for CI, which is not sized but which invalidates 32-byte cache blocks. The cache-coherency protocol is a "write invalidate" protocol, where the cache being written issues a CI transaction if the line is not exclusive. This indicates to all caches that they should immediately invalidate the line, since it will contain "stale data" after the write completes. All caches "snoop" CR transactions and assert MSH if the address of the transaction is present in their cache. Observing MSH, caches can update the state of the lines they hold. If a cache is the "owner," it asserts the signal MIH to tell memory not to send data and then supplies the data to the requesting cache. CRI and CWI are simply the combination of a CI transaction with either a CR or Write transaction. Their purpose is to reduce the quantity of MBus transactions needed and thus conserve bandwidth.

#### 17.4.1 Timing of MSH, MIH, and MRDY

Different cache designs require differing numbers of cycles to complete a snoop request and act on it by asserting MSH and MIH. For proper operation, the memory controller must not acknowledge any data, while MIH might be asserted on CR and CRI transactions.

The memory controllers of systems designed to operate with several types of processors will need programmable timing for the minimum MRDY response time for CR transactions. The minimum MRDY timing should be variable from A+2 to at least A+7 to accommodate various processor modules. The programmable minimum should apply to all CR and CRI transactions.

The SuperSPARC processor requires that MRDY occur on or after the A+3 cycle for all transactions in the direct MBus configuration. The MXCC requires that MRDY occur on or after A+7 in asynchronous clocking configurations and A+5 in synchronous clocking configurations.

At system startup, memory controller MRDY timing should be set to match the minimum MRDY timing of system's slowest snooping cache. The timing of all coherent caches need not be the same within a system.

The MBus specification allows for a minimum cycle time for a read transaction of two cycles (during a read cycle with an error acknowledgment asserted where no data will be provided). Some processor modules will not behave correctly with two cycle transactions. Proper operation with all processors can be assured by delaying the generation of MRDY, MRTY, and MERR to A+2 to be compatible with normal read cycle timing. SuperSPARC and the MXCC require that all transactions end on or after A+2.

### 17.4.2 Module ID

All MBus slave interfaces, including SuperSPARC and the MXCC, accept an (ID[3:0]) input that is used as an aid to system configuration. As a Level 2 master, SuperSPARC and the MXCC also use the values on ID[3:0] as output (MAD[63:0]) during the address phase of every transaction. The processor's module number is determined at reset by sampling the MID [3:0] pins. The MID[3:0] signals are connected to SuperSPARC address bus, pins ADDR[3:0]. The module number should be asserted statically by system hardware. The number may be any value except 0000. A module number of 0000 references the reserved boot mode address space.

### 17.4.3 Wrapping

A wrapped transaction transfers the requested word first and then the rest of the requested block in sequential order, wrapping from the last word of the block to the first.

Wrapping concerns the order in which data will be delivered for multi-cycle transfers. For MBus, Read (with transfer size of larger than eight bytes), CR, and CRI transactions are wrapped transactions. Wrapping implies that the first eight-byte double-word of data to be delivered is specified by the physical address bits during the address phase (MAD[35:3]). The rest of the requested sub-block is to be delivered in sequential order. After delivering the last double-word of the sub-block, the next double-word delivered is wrapped to the first double-word of the sub-block. See Table 17-6.

Table 17-6. Order of Wrapped Bytes

MAD [4:3]	Bytes Returned In			
	First Cycle	Second Cycle	Third Cycle	Fourth Cycle
0	0-7	8-15	16-23	24-31
1	8-15	16-23	24-31	0-7
2	16-23	24-31	0-7	8-15
3	24-31	0-7	8-15	16-23

All SuperSPARC-based modules will issue wrapped requests. Wrapping cannot usually be disabled in a processor. Therefore, memory controllers must support wrapped requests.

A level 2 coherent cache that does not issue wrapped requests may not be able to deliver data to wrapped CR or CRI transactions when they assert MIF. These caches cannot be operated as coherent caches in a system with processors that issue wrapped requests.

#### **17.4.4 Cache-Consistency Protocol**

The level-2 MBus specifies a single-ownership, write-invalidate, cache-consistency policy. Only a single cache may own a modified copy of a cache line at any one time. Multiple shared copies of modified or clean data may exist within the system at any time. These multiple copies are invalidated any time the cache line is modified (the modified data may then be re-read from the owner and cached as modified and shared information). This protocol is described in the MBus specification.

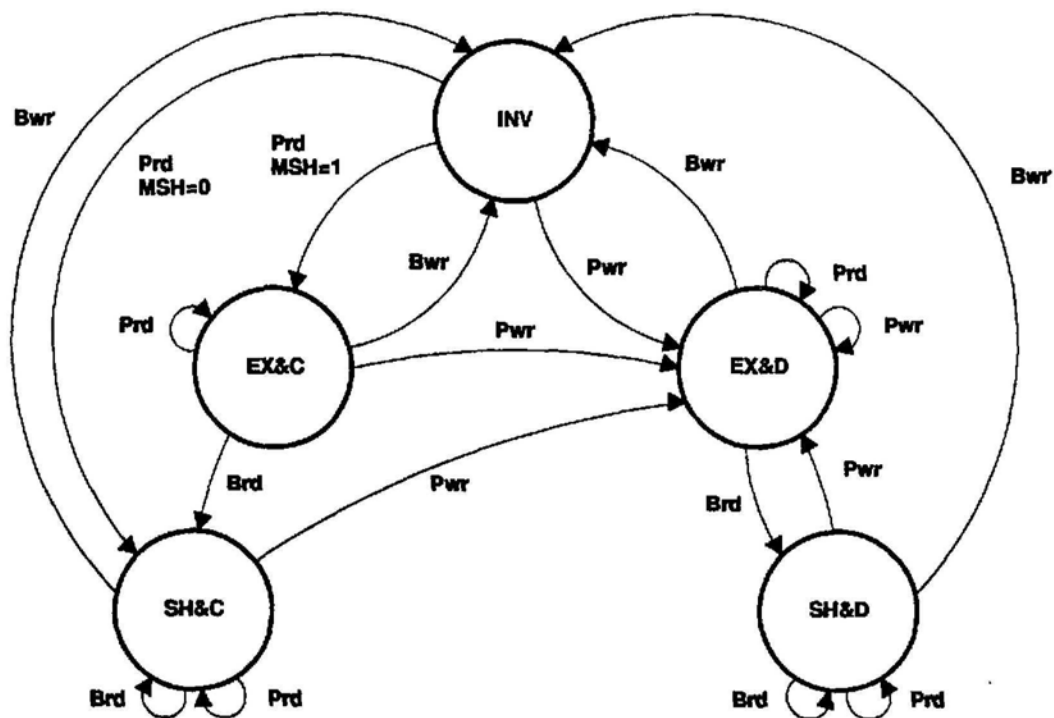
During any CR transaction, all processors in the system will assert the *MSH* (Shared) signal if they currently have a cached copy of that data. Since several caches may be sharing the information, the *MSH* signal is open drain and can be asserted by all caches simultaneously. *MSH* must be pulled inactive (high) external to any processor (typically a resistive pull-up). *MSH* may be asserted at any time prior to the first data acknowledgment for the transaction. *MSH* must be asserted for at least one cycle.

If a cache owns the data for the transaction, it may intervene in a read transaction by asserting the *MIH* (memory inhibit) signal. Since there may be only one owner at a time for each cache line, only a single cache may assert *MIH*. See Subsection 17.4.1 and Section 17.8 for information on the timing of the *MSH* and *MIH*.

#### ***Cache Consistency in Direct MBus Configuration***

Once the SSP asserts *MIH* in response to a read (CR or CRI) transaction, it will complete the bus cycle by supplying data and a ready reply starting four cycles later. During these four cycles, the memory controller can drive other data and ready signals in response to the transaction for two cycles, followed by a third cycle where the bus is driven high. Any replies received during this time are ignored by the SSP. No exceptions may be reported after asserting *MIH*. Figure 17-5 represents the cache-consistency algorithm used by SuperSPARC's data cache on the MBus.

Figure 17-5. MBus Cache Consistency State Diagram for Data Cache

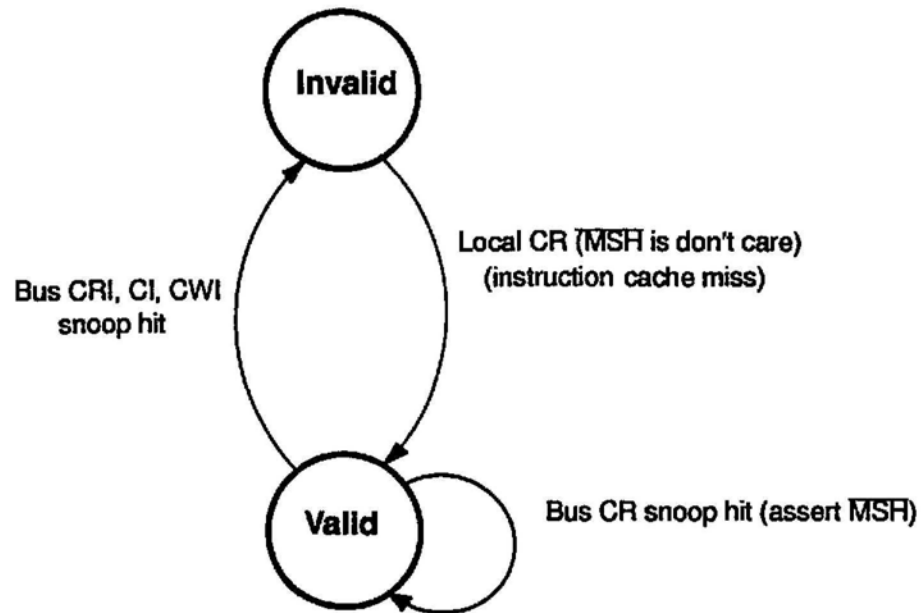


The following symbols are used in the diagram:

- Prd Local read
- Brd System bus read (CR) snoop hits
- Pwr Local write
- Bwr System bus invalidate (CI, CRI or CWI) snoop hits

Figure 17-6 represents a simpler consistency algorithm used by Super-SPARC's instruction cache on the MBus.

Figure 17-6. MBus Cache Consistency State Diagram For Instruction Cache



#### 17.4.5 MultiCache Controller Consistency in Full Module Configuration

MBus cache consistency in the full module configuration is similar to that of the direct MBus configuration.

The ownership of a sub-block belongs to the external cache of the processor that last wrote to the sub-block. A sub-block remains owned until another cache attempts to write the sub-block or the sub-block is copied back to memory (usually when it is replaced in cache).

Ownership is obtained or transferred from cache to another through write invalidation. When an SSP issues a write to a sub-block that is shared ( $S=1$ ), the MXCC issues a (CI) transaction to MBus and assumes ownership of the sub-block. When the CI transaction is received by other caches on MBus, they invalidate the copies that they have of the sub-block. If the write is to a sub-block that is not present in the E-cache (write miss), the MXCC issues a CRI transaction on the bus. If another cache owns the sub-block, it supplies the sub-block and inhibits memory from responding with the MIF signal. After a CRI transaction, the issuer owns the sub-block, and all other caches have invalidated any copies of the sub-block that they previously had.

The cache-consistency state of a sub-block in the external cache may be changed by accesses from the SSP or from MBus. The relationship between the states and accesses from the local processor and MBus are shown in Figure 17-5.

## **17.5 SuperSPARC MBus Transactions (Generic)**

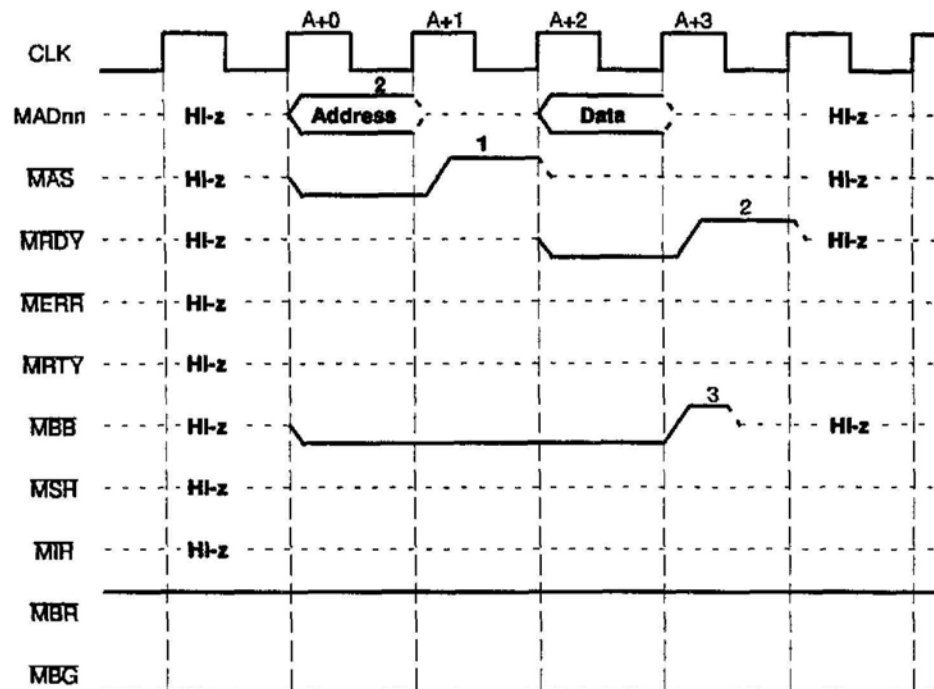
This section provides an outline for generic MBus transactions. Along with an explanation for each transaction, a timing diagram is provided to show the usual order signals are asserted in the transaction. Section 17.6 provides the details specific to a SuperSPARC processor operating in stand-alone mode on the MBus. Section 17.7 provides the details for a SuperSPARC process and the MXCC chip module on the MBus.

### **17.5.1 Read**

Figure 17-7 shows a simple eight-byte read operation. The master drives address and status information on the MAD lines and asserts  $\overline{MAS}$  for one cycle. The addressed slave supplies the data by driving the MAD lines and asserting the  $\overline{MRDY}$  signal for one clock cycle for each 64-bit word transferred.



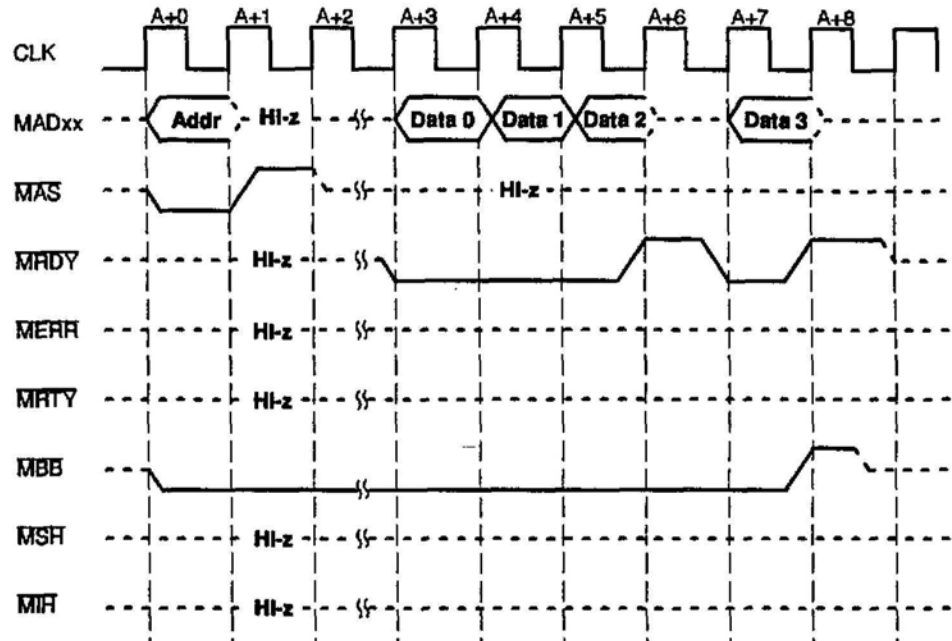
Figure 17-7. MBus Read Cycle



1. Control lines (MAS, MRDY, MERR, MRTY) are driven inactive for one clock before being released
2. MAD(nn) lines are held to their previously driven state by system bus holders.
3. MBB is driven high for 1/2 clock cycle before being released.

Figure 17-8 shows a 32-byte read operation. A read operation can be performed on any size of data transfer that is specified by the SIZE bits. Read transactions support wrapping (critical word first ordering). Transactions involving fewer than eight bytes will have undefined data on the unused bytes.

Figure 17-8. MBus Burst Read Cycle



### 17.5.2 Write

Figure 17-9 shows a simple eight-byte write operation. The master drives address and status information on the MAD lines and asserts **MAS** for one cycle. The Master then drives the data on these same lines, starting at the next cycle. The addressed slave signals completion by asserting the **MRDY** signal for one clock cycle for each eight bytes transferred. Figure 17-10 shows a 32-byte burst transfer.

A write operation can be performed on any size of data transfer specified by the **SIZE** bits. Neither SuperSPARC nor the MXCC will issue a burst operation greater than 32 bytes. Write transactions involving fewer than eight bytes will have undefined data on the unused bytes. The writing master will immediately drive the data in the period after the address phase of the transaction and immediately after the receipt of each **MRDY** in transactions with **SIZE** greater than eight bytes.

Figure 17-9. MBus Eight-Byte Write Operation

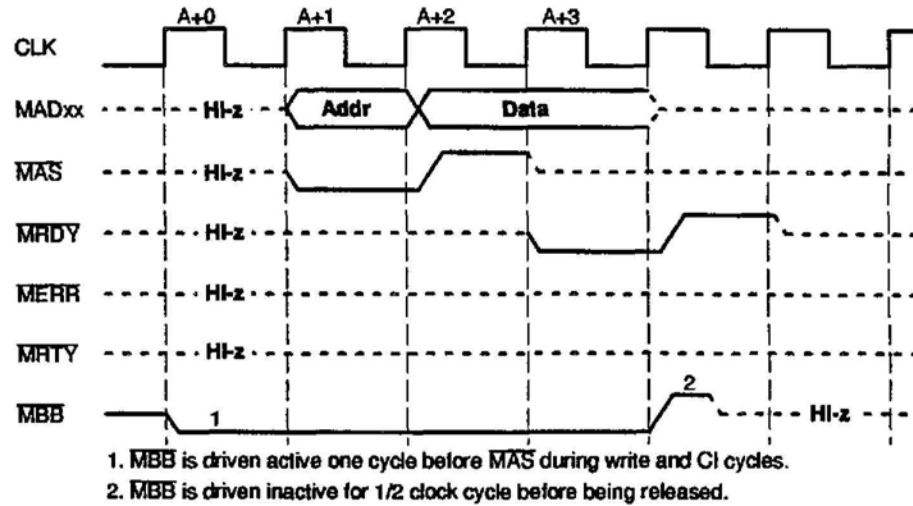
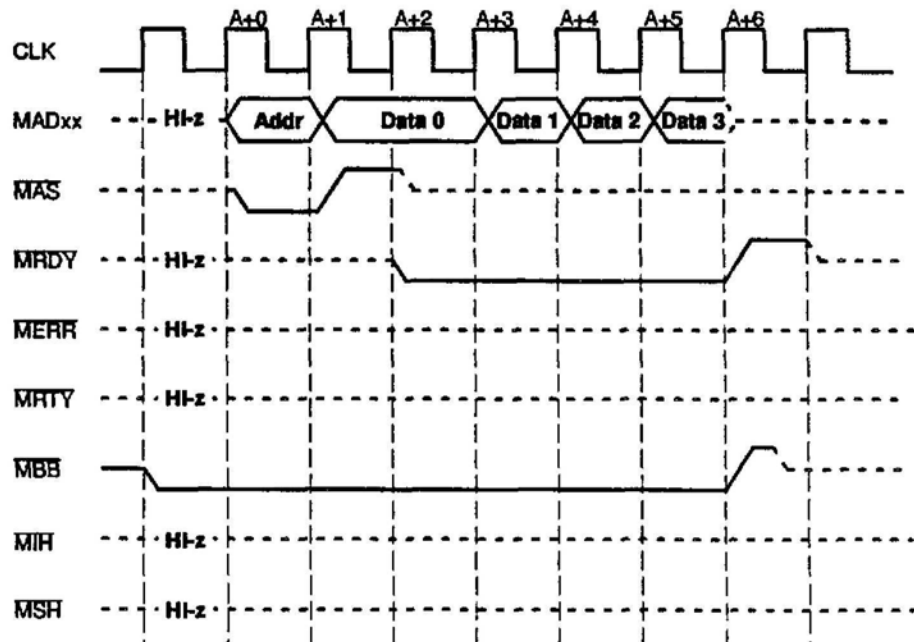


Figure 17-10. MBus 32-Byte Burst Write Operation

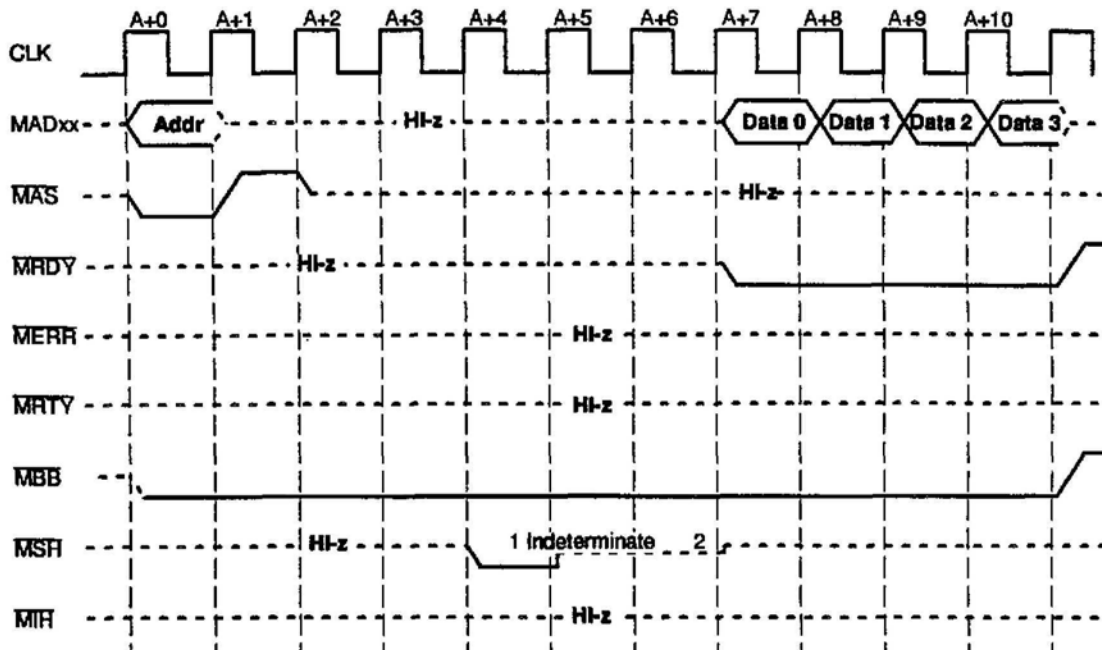


### 17.5.3 Coherent Read

A CR operation is a block read transaction that maintains cache consistency. Coherent reads are performed on data intended to be cached in the SSP's internal and external caches. Participants in the CR transaction are the requesting cache, any other caches that snoop, and memory (or second-level cache). There are three possible read scenarios that the caches that snoop can experience:

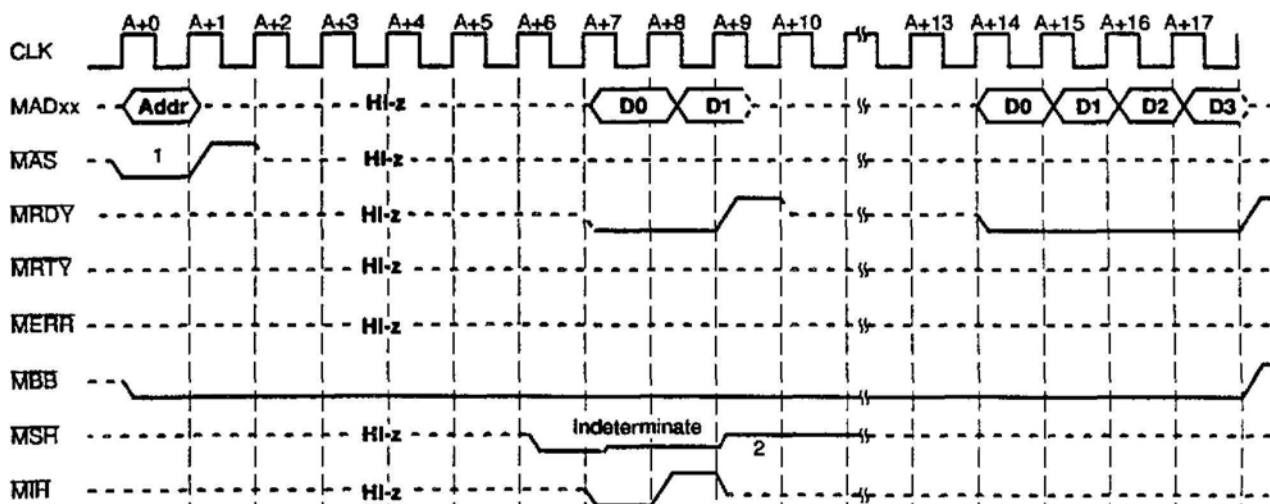
- ☐ For a snooping cache that does not have a copy of the requested block, it simply ignores the transaction.
- ☐ For a snooping cache that does have a copy of the requested block but does not own it, it simply asserts **MSH** for one cycle during the bus transaction (from time **A+2** to time **A+7**). It will mark its copy as shared (if not already marked as such). See Figure 17-11.
- ☐ For a snooping cache that owns the requested block, it will assert both **MSH** and **MIF** for one cycle during the bus transaction (in any cycle from **A+2** to **A+7**) and start supplying the requested data no sooner than four cycles after it issued **MIF**. If its own copy of the block was labeled exclusive, it will be changed to shared; otherwise, no status change will take place for its copy. Figure 17-12 shows a cycle in which a cache owns the data and supplies it to the requesting master.

Figure 17–11. MBus Coherent Read of Shared Data



1. MSH may occur in any cycle from A+2 to A+7.
2. MSH is an open drain signal. It is not driven inactive. The System pull-up resistor returns it to an inactive level.

Figure 17–12. MBus Coherent Read of Owned Data



1. Device is not the Master.
2. MSH is an open drain signal. It is not driven inactive. The System pull-up resistor returns it to an inactive level.

On receiving the data block, the requesting master will label the block exclusive if no **MSH** signal was asserted during the bus transaction. The earliest that a slave is allowed to issue **MRDY** for a CR operation is system-dependent (see Subsection 17.4.1 and Section 17.8). This ensures that acknowledgments never occur after **MIF** (concurrent **MIF** and acknowledgments are OK).

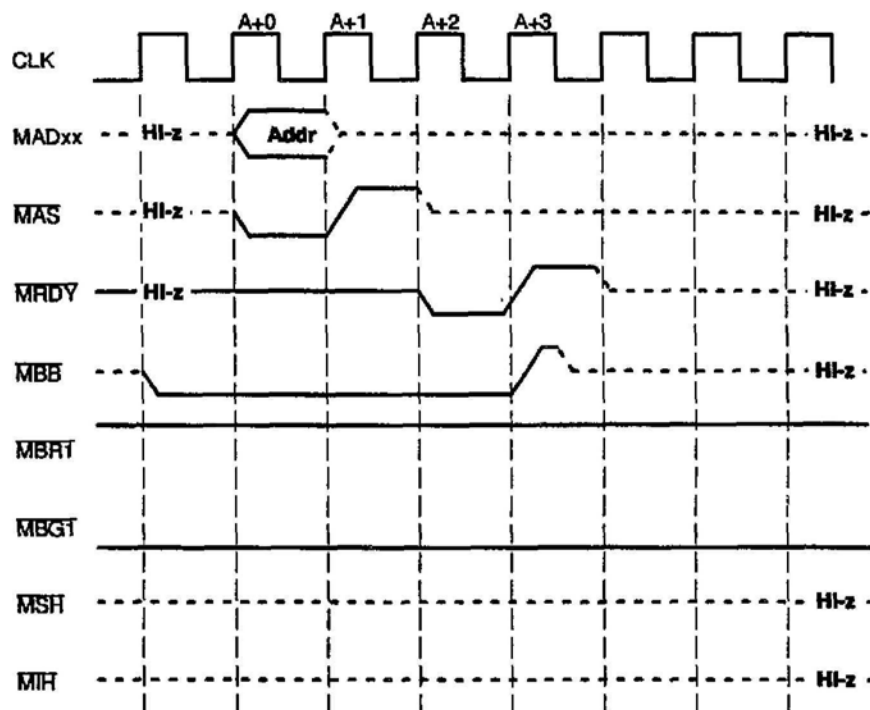
#### 17.5.4 Coherent Invalidate

An Invalidate operation can only be performed on a block (32 bytes). All Invalidate operations will be snooped by all snooping caches. If an Invalidate operation hits in a cache, that copy will be invalidated immediately, regardless of its state. Memory is responsible for the acknowledgment of the CI transaction. This is accomplished on cycle A+2 or later. All acknowledgment types except error acknowledgments are possible. Memory will only issue normal acknowledgments to CI transactions, but third-level caches may issue other acknowledgments, particularly R&R. It should also be noted that a CI transaction will have **SIZE = 32B** during the address phase, but will only be expecting one **MRDY** for the acknowledgment. If, in a particular system, caches cannot guarantee to complete their invalidation before their A+2 cycle, the memory controller for that system should delay the acknowledgment as appropriate.

CI MBus transactions are issued when a write is being performed into a shared cache block. Before the write can actually be performed, all the other system's caches must have their local copies invalidated (write-invalidate cache-consistency protocol). Snooping caches need not assert **MSH**. The **MAD[63:00]** lines will contain undefined data during the data phase cycles. Figure 17-13 shows a CI operation.

If a CI transaction should receive an R&R acknowledgment, there is a possibility that the block that is about to be written could become invalidated by an intervening invalidation transaction on the bus. This means that, when the cache regains the bus, it should issue a CRI transaction, not a CI transaction, to once again allocate the (sub-) block. Figure 17-13 shows a CI operation.

Figure 17-13. MBus Coherent Invalidate Operation



### 17.5.5 Coherent Read and Invalidate

Since the MBus supports a write-invalidate type of cache-consistency protocol, a special CRI transaction that combines a CR transaction with a CI transaction was included to reduce the number of MBus Coherent transactions. Caches that are performing CR transactions with the knowledge that they intend to immediately modify the data can issue this transaction.

Each CRI transaction will be snooped by all system caches. If the address hits and the cache does not own the block, that cache will immediately invalidate its copy of this block, no matter what state the data was in. If the address hits and the cache owns the block, the block will assert MIF and supply the data. When the data has been successfully supplied, the cache will then invalidate its copy of this block.

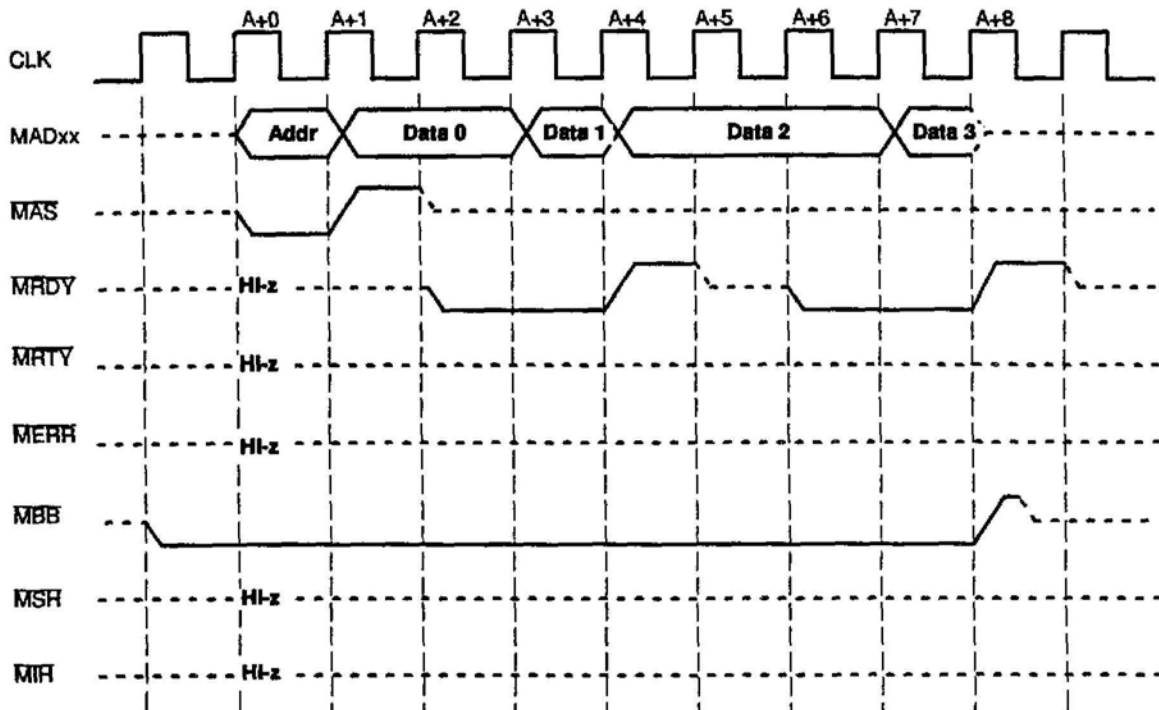
MSH is not driven during the CRI transaction.

### 17.5.6 Coherent Write and Invalidate

A Coherent Write and Invalidate transaction combines a block write transaction with a CI transaction. Figure 17-14 shows a Coherent Write and Invalidate operation.

Each Coherent Write and Invalidate transaction will be snooped by all system caches. If the address hits, caches will invalidate their copies of this block, no matter what state the data was in. Neither MIF nor MSH is asserted for Coherent Write and Invalidate transactions.

Figure 17-14. MBus Coherent Write and Invalidate



### 17.5.7 Acknowledgment Cycles

Any transaction, once issued, must correctly accept any acknowledgment type. The earliest that an acknowledgment will be issued is system-dependent (see Subsection 17.4.1 and Section 17.8). Caches that are supplying data as part of a CR transaction may only issue either normal or error acknowledgments. They may not issue R&R or Retry acknowledgments. Table 17-7 shows the valid acknowledgments that a slave may issue.



Table 17-7. Error and Retry Handling

MERR	MRDY	MRTY	Description
H	H	H	Idle Cycle
H	H	L	Relinquish and Retry
H	L	H	Valid Data Transfer
H	L	L	RESERVED
L	H	H	Bus Error (ERROR1)
L	H	L	Timeout Error (ERROR2)
L	L	H	Uncorrectable Error (ERROR3)
L	L	L	Retry

### Idle Cycle

When there is no bus activity or when it is necessary to insert wait states between the address cycle and the data cycle or between consecutive data cycles, an addressed slave can simply refrain from asserting any transaction acknowledgment types (MERR, MRDY, and MRTY). The number of wait cycles that can be inserted is arbitrary, as long as the number does not exceed the system timeout interval.

### Relinquish & Retry (R&R)

When a slave device cannot accept or supply data immediately, it can perform a Relinquish and Retry (R&R) acknowledgment cycle by asserting MRTY for only one bus cycle. This will indicate to the requesting master that it should release the bus immediately so that the bus can be re-arbitrated and possibly used by another master. This release of the bus will provide at least one dead cycle before the transaction can be retried. When a master that receives an R&R regains bus mastership, it must re-issue the same transaction from the beginning. An exception to this is when a CI turns into a CRI. R&R can only be issued on the first acknowledgement of a transaction. It is the responsibility of the slave port to time the duration of the transaction that is causing it to issue R&R and return an ERROR2 acknowledgment to the correct master when its device-specific timeout interval has passed and the master has re-connected to the slave.

There are two different cases in which slaves issue R&R acknowledgments:

- ☐ The first case is a slow device. If a device is slow to respond, the slave interface should wait a short interval (about one microsecond is recommended), and then issue an R&R acknowledgment. It should also capture the ID of the master (MID[3:0]) from the MAD lines during the address phase and enter a "port busy" state while waiting for the device attached to the slave to respond. The master will eventually reconnect, and the R&R process will be repeated until either the device responds or the slave time-out interval is exceeded. The slave will then issue the normal or error acknowledgment, respectively, and exit the "port busy" state.

If a master with an ID other than that captured by the slave port should access the slave port while it is in the "port busy" state, it should simply be given an R&R acknowledgment.

- ☐ The second case for R&R acknowledgments is the resolution of deadlock situations where there is master and slave port sharing an MBus interface and simultaneous transactions on both ports require one transaction to back off. R&R requires that the current owner of MBus relinquish ownership in order to resolve the deadlock. R&Rs used to resolve deadlocks are inherently stateless and do not require a "port busy" state.

A detail of significance is that R&R can be issued to a transaction that is part of a locked sequence of transactions. By definition, transactions in a locked sequence are addressed to the same device (e.g., main memory (or second-level cache) or an I/O adapter. There is only one "port busy" state per device, so there is only one source of R&R for a locked sequence.

It should be noted that caches that assert MIF and then supply data are not permitted to issue R&R acknowledgements. Figure 17-15 and Figure 17-16 show R&R acknowledgments.

Figure 17-15. Relinquish and Retry (Parked)

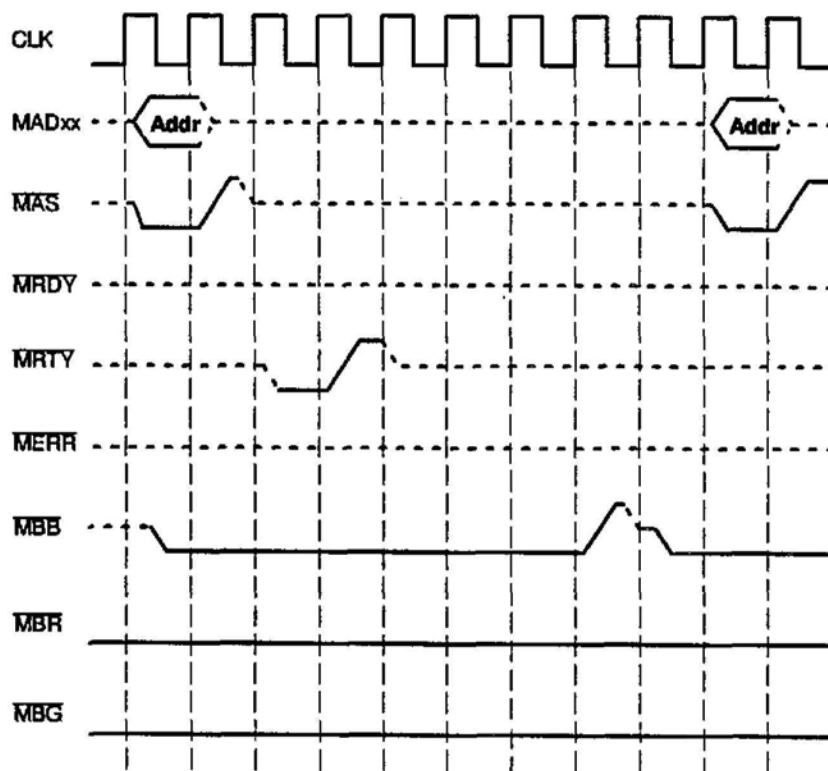
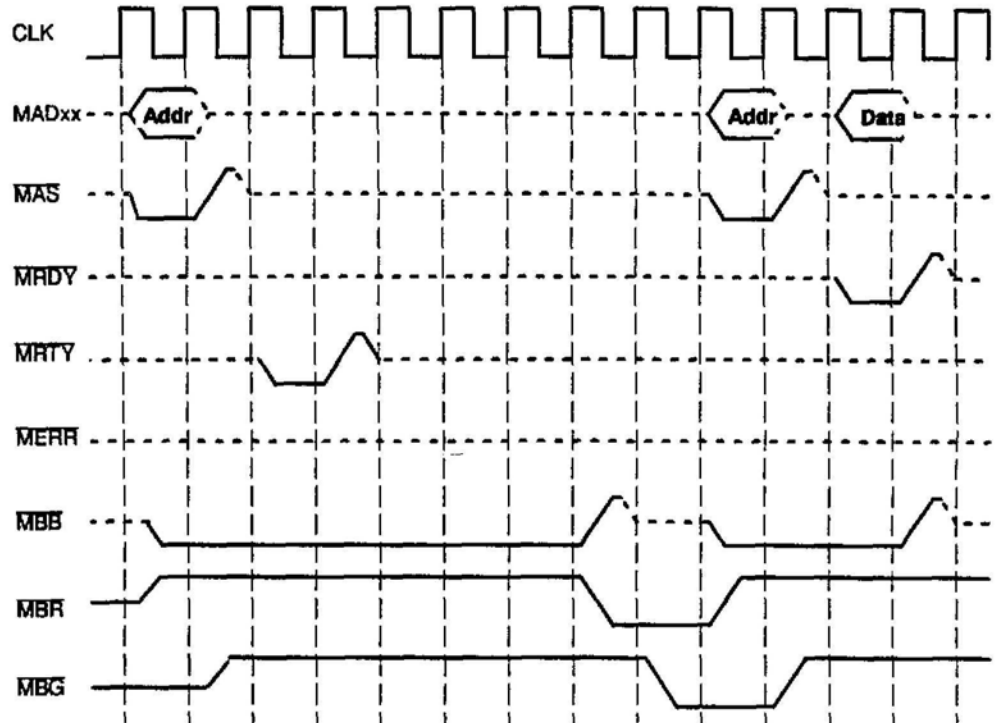


Figure 17-16. Relinquish and Retry (Rearbitrate).

**Valid Data Transfer**

A valid or ready data transfer is indicated by a responding slave with the assertion of the **MRDY** transaction status bit.

**Error1 (Bus Error)**

When the responding device asserts only the **MERR** transaction status bit, the requesting master will interpret this as an external bus error. This error code can also be used to indicate the same system implementation dependent error. Bus error is the suggested interpretation of an **ERROR1** acknowledgment.

**ERROR2 (Timeout)**

This acknowledgment is expected to be generated by some sort of watchdog logic in the system, after a set number of cycles have elapsed with the **MBB** signal asserted. The number of cycles for the timeout interval is system implementation-dependent. An interval of 200 microseconds is recommended. This error code can also be used to indicate a system implementation-dependent error. Timeout is the suggested interpretation of an **ERROR2** acknowledgment.

### ***ERROR3 (Uncorrectable)***

This acknowledgment is used mainly by the addressed memory controller to inform the requester that, in the process of accessing the data, some sort of uncorrectable error has been encountered (such as parity, uncorrectable ECC, etc.). This error code can also be used to indicate a system implementation-dependent error. Uncorrectable error is the suggested interpretation of an ERROR3 acknowledgment.

### ***Retry***

This acknowledgment causes the master to restart the transaction from the MAS cycle without releasing the bus for re-arbitration. This differs from Relinquish and Retry in that it does not allow any other master to access the bus before the retry.

---

**Note:**

This acknowledgment should not be used with MXCC. See Subsection 17.6.9.

---

## 17.6 MBus for SuperSPARC Processor Without the MXCC

The following section deals with information specific to a SuperSPARC processor connected directly to the MBus with no MXCC (the direct MBus configuration in Table 17-1 and Figure 17-1). For details on MBus behavior of the SuperSPARC Processor with the MXCC, see Section 17.7.

### 17.6.1 Compatibility

The SSP can interoperate on MBus with some other MBus processor modules, but not all. In particular, modules that require the virtual address superset bits will not be compatible. The SSP responds properly to all standard MBus transactions, though it does not produce all of them. The only size of burst transactions supported by the SSP are 32-byte wrapped bursts. All cache-consistent transactions operate on 32-byte sub-blocks. All responses on *MSH* and *MIF* are provided in the A+3 cycle (the third cycle after assertion of address on the bus). SuperSPARC can operate in a system with variable response times, up to A+7. MBus systems should be designed to accept these timing variations. Even though SuperSPARC may be protocol-compatible with some modules, electrical and signal timing differences may make certain modules incompatible. See the SuperSPARC data sheet for pin-timing details.

### 17.6.2 Selecting MBus

MBus operation is selected by driving the *CCMODE* pin inactive. This signal should be driven statically by the system. Any change in the state of this signal after the deassertion of *RESET* will result in unpredictable operation. Software may determine the state of *CCMODE*, and the direct MBus interface is therefore selected by reading the *MCNTL.mb* bit.

### 17.6.3 Port Register

The MBus port register contains vendor identification information for the component. Processors respond to Read transactions to an address that is based on the current module ID value. See the MBus specification for further details. SuperSPARC will return the value 0x00000004 on *MAD[31:0]* in response to any read of its port register. This indicates device 0, revision 0, for the vendor (Texas Instruments). The version number may change in future releases of the component.

### 17.6.4 SuperSPARC Processor as MBus Master

#### *Read*

The Read transaction is used for read operations of both code and data that will not be snooped by other caches (non-cacheable). All non-cacheable reads will cause the store buffer to drain before the transaction begins.

Only single-cycle (64 bits maximum) transfers will be used. SuperSPARC does not burst transfer reads. The transaction may be byte, half-word, word, or double-word in length. Big-endian word ordering is used (byte 0 appears on the high order bits of the bus).

The A+2 cycle is the minimum cycle that SuperSPARC can accept an acknowledgement for a Read.

## **Write**

Non-cacheable stores are queued in SuperSPARC's store buffer, when it is enabled (MCNTL.SB), and SuperSPARC does not wait for the stores to complete.

As soon as SuperSPARC receives a bus grant, the queued non-cacheable store is issued to the bus as a write transaction. Burst writes are not used for non-cacheable writes. Write transactions may be byte, half-word, word, or double-word in size. Any errors on the write transaction are reported as deferred data store errors. For synchronization, any cacheable store waits until the store buffer has completed all non-cacheable stores before beginning.

The A+2 cycle is the earliest cycle on which SuperSPARC can accept an acknowledgement for a Write.

---

### **Note: Atomic Transactions**

Non-cacheable atomic transactions are caused by execution of either SWAP or LDSTUB instructions. These operations are performed as a locked sequence of read and write transactions. The lock bit field in the address phase of the read and write transactions will be set. The SSP will not release the bus between the two transactions unless explicitly requested by a relinquish & retry (R&R) reply. R&R replies are accepted for both read and write portions of atomic transactions.

---

---

### **Note: Page-Table Walk Operations**

All page tables should be treated as non-cacheable in SuperSPARC direct MBus systems. As a result, only Level-1 transactions are used to reference page tables. During a table walk, SuperSPARC maintains bus ownership, unless explicitly told to release by an R&R response. The lock bit in the MAD field is set. All levels of the page table are read with standard single-word read transactions. Updates to R and M bits, or just M bits, are done with standard write operations. Updates to the R bit only will be done with non-cacheable atomic swaps (see non-cacheable atomic references, above). This mechanism eliminates potential status bit inconsistencies when multiple MMUs are attempting to update page tables simultaneously. See Section 9.4.

---

### **Coherent Reads**

CR transactions are used to read data from the current owner. The owner may be main memory or another cache. CR transactions will be used for all data cache load misses and for all instruction cache misses. If the data is owned by another cache, it will respond by asserting **MIF** and providing the data.

All CR transactions use wrapping (see Subsection 17.4.3), and the starting address specified is the first double-word transferred. Once the needed data arrives, the processor uses it immediately.

Any processor or cache that has a valid cached copy of the data referenced by the CR transaction must assert the **MSH** signal to indicate that data is shared. SuperSPARC can accept the assertion of the **MSH** signal at any time up to or concurrent with the receipt of the first data word.

If the data is owned by another cache, that cache asserts **MIF**. The SSP ignores any data-ready (**MRDY**) responses until four cycles beyond the assertion of **MIF**. This allows memory controllers to begin transmitting data sooner than they might have otherwise. Memory controllers should not respond with data until a time equal to the longest **MIF** delay of any cache in the system.

The SSP can accept **MIF** up to and concurrent with the first **MRDY**. The earliest that an acknowledgement can be accepted is at A+2, as long as there has been no **MIF**. If **MIF** is asserted by any cache after **MRDY** has been asserted, the resulting behavior will be unpredictable.

### **Coherent Read and Invalidate**

CRI transactions are used to read data from the current owner for write-allocate operations. Use of these transactions implies that the data will be modified upon arrival at the processor. The current owner should relinquish ownership, and all copies of the cache block should be invalidated. After the transaction completes, the SSP issuing the CRI transaction is the exclusive owner of the cache block.

CRI behaves similarly to the CR transaction with respect to wrapping and timing issues.

### **Coherent Write and Invalidate**

The SSP does not issue the CWI transaction. MBus does not provide any true Coherent Write (CW) transactions. Due to the write-allocate scheme used by the SSP's caches, a processor must own data before it may write to it. As a result, all cacheable write operations are done locally within a processor cache. The Write transaction is used for all non-cacheable stores and for cache copy-back operations. The SSP will relinquish ownership of a cache block after a copy-back operation (that uses a Write transaction).



### ***Coherent Invalidate***

The SSP requests only a single type of pure-consistency operation (operations that transfer no data but affect the state of a cached block). This is a CI transaction. A CI transaction will invalidate all cached copies of a block in the system. This is used when the processor attempts a store to a shared line, regardless of ownership. After the CI transaction, the issuing processor becomes the exclusive owner of the cache block. The processor waits for proper completion of the CI transaction before performing the internal write.

The SSP expects an acknowledgement, if one is sent, at A+2. The processor can in fact accept an acknowledgement at A+1, but systems normally cannot generate acknowledgements that quickly.

### **17.6.5 SuperSPARC as a Snooping Cache Directly on MBus**

The SSP maintains cache-consistency by snooping on the MBus. The non-cacheable read and write transactions are ignored by SuperSPARC as a cache snoop. Therefore, the relevant transactions are all coherent transactions.

### ***Coherent Read***

The SSP responds with the proper *MSH* and *MIF* signals in response to a CR transaction on the bus. These signals are driven in the A+3 cycle.

If the SSP owns the cache block, it waits an additional four cycles after asserting *MIF*, until A+7, then responds with the data from its internal data cache. The *MSH* pin is driven as an open drain signal. It is only driven low, and it must be pulled inactive externally. Anytime *MIF* is driven, it is driven low for a cycle, then driven high again for a cycle; the driver then returns to high impedance.

### ***Coherent Read and Invalidate***

The SSP responds to a CRI transaction snoop hit in a manner similar to the CR transaction. The timing remains the same as that for a CR transaction snoop hit (A+3 for *MIF* and A+7 for acknowledgement and data), but *MSH* will never be asserted in a CRI transaction. *MIF* will be asserted if the processor owns the data.

**Note:**

Four cycles are required between asserting of **MIH** and **MRDY** for CR and CRI transactions because of the following timing:

cycle 1: Owner asserts **MIH**, Memory Controller asserts **MRDY**

cycle 2: Memory Controller asserts 2nd **MRDY** and "sees" the Owner's **MIH**

cycle 3: Memory Controller starts disable

cycle 4: Three-state

**Coherent Write and Invalidate**

CWI is a combined transaction that has the effect of a Write transaction followed immediately by a CI transaction. Since write transactions are non-coherent, the SSP does not snoop these operations. Coherent Writes and Invalidate transactions are treated as CI transactions. The SSP can accept these every other bus cycle and no acknowledgement is required.

**Coherent Invalidate**

The SSP treats the two pure consistency operations, where no data is requested, as simple invalidations. The two pure consistency transactions are CI and CWI. Since no reply from the processor is required, SuperSPARC can accept CI transactions at maximum bus rate, every other cycle. The rate of these transactions is controlled by system logic and will generally be slower than two cycles per transaction.

**17.6.6 SuperSPARC Processor as MBus Slave**

The SSP behaves as a slave only when its port register is read. Only Read transactions are supported to the port register. All Write, CR, CRI, CWI, and CI transactions are ignored by an SSP slave.

**Read**

MBus port registers may be read using non-cacheable accesses to physical addresses in the range 0xff100000->0xffff0000, depending on the module number being addressed. In this case, the SSP will behave as a slave and respond in minimum of A+3 cycles.

It is legal for a processor to address its own port register. This is the only case of snooping on non-coherent read transactions. In this case, the processor in fact acts as both master and slave on the same transaction.

**17.6.7 Store Buffer Operation in Direct MBus Configuration**

The SSP's store buffer will be used to buffer all copy-back data, as well as non-cacheable stores in direct MBus configuration. Any errors on these transactions will be reported as deferred data\_store\_errors.

The SSP's store buffer maintains consistency. During copy-back operations, the store buffer becomes the owner of a cache block. The store buffer snoops all coherent operations and responds appropriately with the status of the block. The contents of the copy-back buffer are always owned. Data is always returned in critical word first order. Table 17-8 describes the actions taken if a coherent transaction hits copy-back data in the SSP's store buffer.

Table 17-8. Store Buffer Copy-Back Snoop Hit Actions

MBus Transaction	Transaction
CR	MIF, MSH, supply data, continue copy-back
CRI	MIF, supply data, cancel copy-back
CI	cancel copy-back
CWI	cancel copy-back

### 17.6.8 Bus Arbitration

The SSP requests the use of the MBus with the  $\overline{\text{MBR}}$  signal. This signal is asserted when any internal bus cycle is pending. Since the SSP issues some transactions speculatively, the request for a particular transaction may disappear after the bus has been requested. If this occurs, the  $\overline{\text{MBR}}$  signal is deasserted immediately. If the internal request is still valid when the MBus is granted, the transaction occurs. The SSP attempts to overlap arbitration with current bus cycles (including its own bus cycles). A processor is granted future use of the bus by receiving the  $\overline{\text{MBG}}$  signal. When  $\overline{\text{MBG}}$  is received, the bus may still be busy servicing the previous owner. This is indicated by the  $\overline{\text{MBB}}$  (MBus busy) signal. The  $\overline{\text{MBR}}$  signal is deasserted as soon as  $\overline{\text{MBG}}$  is asserted. In order to gain ownership of the bus, the processor waits for its  $\overline{\text{MBG}}$  signal to be active and the  $\overline{\text{MBB}}$  to be deasserted. Once this occurs, it asserts  $\overline{\text{MBB}}$ . After arbitration is complete, the processor asserts  $\overline{\text{MAS}}$  to begin the transaction. Bus busy is maintained during locked and atomic transactions (like table walks). If the SSP already owns the bus, there is no arbitration delay to begin subsequent transactions.

### 17.6.9 Error and Retry Handling

Several different transaction responses are defined by the MBus. Successful bus cycles are terminated with the valid data transfer acknowledgment (MRDY). Unsuccessful transactions may be terminated with several different error responses, or retries. The SSP supports all error responses but only one of the retry responses. All error responses are applied to the current data transfer only. Any data received with a valid data transfer response will be assumed correct and used internally. If any errors occur during the transfer of a cache block, the internal cache will not be validated. All data returned prior to the error may be used internally. See Table 17-9.

Table 17-9. Error and Retry Handling

MERR	MRDY	MRTY	Description
H	H	H	Idle Cycle
H	H	L	Relinquish and Retry
H	L	H	Valid Data Transfer
H	L	L	RESERVED
L	H	H	Bus Error (ERROR1)
L	H	L	Timeout Error (ERROR2)
L	L	H	Uncorrectable Error (ERROR3)
L	L	L	Retry†

† Guaranteed by design but not tested.

### Errors

Three error responses are defined. These are ERROR1 (Bus Error), ERROR2 (Timeout), and ERROR3 (Uncorrectable). The SSP treats all these errors in the same manner and sets the MFSR to indicate the exact error response, as shown in Table 17-10.

Table 17-10. MFSR Response to Bus Errors

Response	MFSR Setting
ERROR1 (BUS ERROR)	MFSR.BE
ERROR2 (TIMEOUT)	MFSR.TO
ERROR3 (UNCORRECTABLE)	MFSR.UC

### Asynchronous Errors

The SSP asserts the  $\overline{\text{AERR}}$  signal whenever the processor enters error mode or whenever an exception occurs during a store buffer copy-out.  $\overline{\text{AERR}}$  will remain asserted until the MFSR register is cleared of the exception (it is cleared on read).

### ***Relinquish & Retry (R&R)***

The SSP supports (R&R) replies only on the first response to any transaction, including copy-back operations. After receiving an R&R, SuperSPARC will release bus ownership and attempt to retry the transaction.

### ***Retry***

SuperSPARC does not support the retry reply. All retries will be returned in the equivalent of ERROR3 replies. SuperSPARC expects that all data returned to the processor with a valid data transfer (MRDY)READY response is truly correct. This data is used immediately by the pipeline. SuperSPARC cannot tolerate late error responses under any circumstances.

## **17.7 SuperSPARC Processor With the MultiCache Controller on MBus**

The following section deals with information specific to the SSP and the MXCC module connected to the MBus. This is the Full Module MBus configuration in Table 17-1 and Figure 17-2. For details on SuperSPARC's behavior when it is connected to the MBus without the MXCC, see Section 17.6.

### **17.7.1 E-Cache**

The MXCC controls SuperSPARC's external cache memory (E-cache). E-cache is a direct-mapped and copy-back cache, 1M-byte in size in MBus configurations. In MBus configuration, the sub-block size is 32 bytes. There are four sub-blocks per cache block, each block thus containing 128 bytes.

E-cache is a unified cache; it combines instructions and data in a single cache. E-cache maintains the inclusion property with respect to the SSP's on-chip data and instruction caches; every cache block in either of the SSP's caches is also in E-cache.

Should the MXCC be connected to E-cache, startup software should initialize the cache before enabling it by setting the cache enable (CE) bit in the MXCC's configuration register. When the E-cache is disabled, *MSH* and *MIF* are always deasserted, and normal cacheable accesses bypass the E-cache. In this case, the MXCC passes CI, CRI, and CWI transaction snoop requests on to the SSP to enable the processor to invalidate its on-chip cache.

### **17.7.2 Selecting MBus**

MBus operation is selected by driving the MBSEL pin on MXCC high. This signal should be driven statically by the system. The SSP's *CCMODE* pin must be held low to select the processor's VBus interface to MXCC. Any change in the state of these signals during the operation of the devices will result in unpredictable operation.

### **17.7.3 Port Register**

The MBus port register contains vendor identification information for the component. Processors respond to read transactions to an address that is based on the current module ID value. See the MBus specification for further details. The MXCC returns the value 0x00000104 on MAD[31:0] in response to any read of its port register. This indicates device 1, revision 0 for the vendor Texas Instruments (vendor number 4). The version number may change in future releases of the component.

### 17.7.4 Synchronous and Asynchronous Clocking

The MXCC can be clocked in two different fashions. Asynchronous clocking results if PCLK (clock on the processor side of the MXCC) is faster than BCLK (clock on the bus side of the MXCC). Due to the design of the internal synchronizers, PCLK must, in asynchronous mode, be at least 300 KHz faster than BCLK, and the ratio of PCLK to BCLK must not exceed 3 to 1.

In synchronous mode, PCLK and BCLK must be connected to the same clock, and there should be very little skew between the two clock inputs (no more than 150ps). See Section 23.3.

### 17.7.5 MXCC Master on MBus

#### **Read**

The read transaction is used for read operations of both code and data that will not be snooped by other caches (non-cacheable). Reads can be performed on any size of data transfer, as specified by the SIZE bits. The MXCC will issue no burst operation greater than 32 bytes. Read transactions involving fewer than eight bytes will have undefined data in the unused bytes. Big-endian word ordering is used (the least significant bytes in a word appear on the high bits of the bus).

The A+2 cycle is the minimum cycle in which the MXCC can accept an acknowledgement for a read.

#### **Write**

Writes are snooped (non-cacheable) by other caches. The MXCC issues no burst write transaction greater than 32 bytes. Bytes, half-words, words, and double-words may all be written, with big-endian ordering. Write transactions involving fewer than eight bytes will have undefined data in the unused bytes.

The A+1 cycle is the minimum cycle in which the MXCC can accept an acknowledgement for a Write.

---

#### **Note: Atomic Transactions**

Non-cacheable atomic transactions are caused by execution of either SWAP or LDSTUB instructions on non-cacheable data. These operations are performed as a locked sequence of Read and Write transactions. The lock bit field in the address phase of the read and write transactions will be set. The MXCC will not release the bus between the two transactions unless explicitly requested by an R&R reply. R&R replies are accepted for both read and write portions of atomic transactions.

---

### **Coherent Reads**

CR transactions are used to read data from the current owner. The owner may be main memory or another cache. The participants in a CR transaction are the requesting cache, the snooping caches, and memory. If the data is owned by another cache, it will respond by asserting **MIH** and providing the data. CR transactions are used by the block copy logic to ensure coherency during block copies and by the E-cache controller logic to read data into the E-cache.

All CR transactions use wrapping (see Subsection 17.4.3), and the starting address specified is the first data transferred. Once the needed data arrives, the processor will use it immediately.

Any processor or cache that has a valid cached copy of the data referenced by the CR transaction asserts the **MSH** signal to indicate that the data is shared. With either asynchronous or synchronous clocking, the MXCC can accept the assertion of the **MSH** signal at any time up to and concurrent with the receipt of the first data word.

If the data is owned by another cache, the MXCC in either clocking mode ignores any data-ready (**MRDY**) responses from **MIH** two cycles beyond the assertion of **MIH**. Memory controllers should not respond with data until a time equal to the maximum **MIH** delay for any cache in the system.

The MXCC accepts **MIH** up to and concurrent with the first **MRDY**. **MIH** should never be asserted after **MRDY** by any cache, as the resulting behavior will be unpredictable.

### **Coherent Read and Invalidate**

CRI transactions read data from the current owner. Use of these transactions implies that the data will be modified upon arrival at the MXCC. The current owner should relinquish ownership, and all copies of the cache line should be invalidated. After the transaction completes, the MXCC should be the exclusive owner of the cache line.

CRI transactions behave similarly to CR transactions with respect to wrapping and timing issues.



### **Coherent Write and Invalidate**

CWI transactions combine a block write with a CI. Each CWI will be snooped by all caches. If the Address hits, the caches invalidate their copies of this block, no matter what state the data was in. Coherent Write and Invalidate transactions behave like a Write, except that snooping caches invalidate copies of the block. All sizes are allowed, but only a single 32-byte block is invalidated, regardless of the SIZE specified. Due to the nature of the MBus coherency protocol, neither  $\overline{MIF}$  nor  $\overline{MSH}$  can be asserted. The MXCC can accept the first acknowledgement for a CWI transaction in the A+2 cycle.

CWI transactions are used by the MXCC's block copy logic to ensure coherency during block copies and block zeros.

### **Coherent Invalidate**

The MXCC uses the CI as a pure-consistency operation (operations that transfer no data but affect the state of a cached block). A CI operation invalidates all cached copies of a block in the system. This is used when the MXCC attempts a store to a shared block, regardless of ownership. After the CI operation, the MXCC becomes the exclusive owner of the cache block.

The MXCC expects an acknowledgement at A+2. An acknowledgement for a CI operation is sent by the memory that holds the data. The caches that hold copies do not acknowledge a CI or a CWI operation.

### **17.7.6 MXCC as a Snooping Cache on MBus**

The MXCC maintains cache consistency by snooping on the MBus. The non-cacheable read and write transactions are ignored by the MXCC as a cache snooper. The relevant transactions are thus all coherent transactions.

### **Coherent Read**

The MXCC responds with the proper  $\overline{MSH}$  and  $\overline{MIF}$  signals to cache hits on CR transactions on the bus.

With asynchronous clocking, the MXCC asserts  $\overline{MIF}$  and/or  $\overline{MSH}$  between A+5 and A+7, inclusive. Should the MXCC own the cache block, the acknowledgement and data can come anywhere from A+14 (best case) to A+23 (worst case). If no  $\overline{MIF}$  is asserted, the acknowledgement and data come in cycle A+7 or later.

For synchronous clocking,  $\overline{MIF}$  and/or  $\overline{MSH}$  are asserted at A+5. If the MXCC owns the cache block, the acknowledgement and data can come anywhere from A+16 (best case) to A+23 (worst case). If no  $\overline{MIF}$  is asserted, the acknowledgement and data come in cycle A+5 or later.

### ***Coherent Read and Invalidate***

The MXCC responds to a CRI transaction snoop hit very similarly to a CR transaction snoop hit. The timing remains the same as that of a CR transaction snoop hit (for both asynchronous and synchronous clocking), but *MSH* will never be asserted in a CRI transaction. *MIH* will be asserted if the processor owns the data.

### ***Coherent Write and Invalidate***

CWI transactions are snooped by all system caches. Should an address hit, the data is invalidated, no matter what state it was in. While CWIs are allowed to be of any size, only a single 32-byte block will be invalidated.

The MXCC does not acknowledge CWI. The MXCC can respond to at most one CWI every fourth cycle.

### ***Coherent Invalidate***

The MXCC treats all pure-consistency operations in which no data is requested as simple invalidations. This includes CI and CWI transactions. These invalidations can only be performed on a sub-block (32-bytes) basis.

The MXCC does not acknowledge CI. The MXCC can respond to at most one CI every fourth cycle.

## **17.7.7 MXCC Slave on MBus**

The MXCC contains a number of registers that control the operation of the E-cache, bus, or other MXCC functions, or sense the MXCC status. The MXCC behaves as a slave when these registers are read or written from MBus. All CR, CRI, CWI, and CI transactions are ignored by the MXCC slave—only Read and Write transactions are supported by the MXCC as a slave.

### ***Read***

The MXCC assumes that accesses to these registers are the same as the register size. The MBus SIZE bits are ignored. During a read transaction, the contents of a 32-bit register are driven onto MAD[31:0], a 16-bit register drives its contents onto MAD[15:0], and a 64-bit register drives MAD[63:0]. The unused bits return unpredictable data on a Read.

The MXCC responds to a Read in the A+9 cycle to the A+23 cycle at the latest, depending on the register read.

## Write

The MXCC assumes that Writes to its internal registers are the same as the register size. The MBus size bits are ignored. During a Write transaction, the contents of a 32-bit register should be driven on MAD[31:0], a 16-bit register on MAD[15:0], and a 64-bit register driven on MAD[63:0]. The unused bits are ignored on a Write from the MBus.

The MXCC responds to a Write in the A+9 cycle to the A+23 cycle at the latest, depending on the register written.

---

### Note:

The atomic load-store operation to any of the registers is not supported. A timeout error will be reported should an atomic load-store be attempted.

---

## 17.7.8 Bus Arbitration

MBus arbitration is accomplished by an external arbiter. The actual arbitration algorithm depends on the implementation of the external arbiter. The MXCC asserts **MBR** when it determines that it requires the MBus. It releases **MBR** immediately after receiving **MBG**. **MBG** remains asserted until **MBB** is negated. The MXCC will normally release the **MBB** signal at the termination of the cycle (final acknowledgment); after an error acknowledgment, however, **MBB** remains asserted for a number of cycles while the MXCC completes its internal error processing.

## 17.7.9 Error and Retry Handling

Several different transaction responses are defined by the MBus. Successful bus cycles are terminated with the valid data transfer acknowledgment (**MRDY**). Unsuccessful transactions may be terminated with several different error responses, or retries. As a slave or snooping cache, the MXCC does not drive the **MRTY** signal and can therefore issue only valid data transfer, bus error, and uncorrectable error acknowledgements. As a master, the MXCC decodes acknowledgments according to Table 17-11.

Table 17-11. SuperSPARC With the MXCC Error and Retry Handling

MERR	MRDY	MRTY	Description
H	H	H	Idle Cycle
H	H	L	Relinquish and Retry
H	L	H	Valid Data Transfer
H	L	L	RESERVED
L	H	H	Bus Error (ERROR1)
L	H	L	Timeout Error (ERROR2)
L	L	H	Uncorrectable Error (ERROR3)
L	L	L	Retry

## Errors

Three error responses are defined. These are ERROR1 (Bus Error), ERROR2 (Timeout), and ERROR3 (Uncorrectable). The MXCC master logs these errors in the MXCC error register (CCER) (see Subsection 16.8.1) and, if appropriate, notifies the SSP via a VBus error acknowledgment.

## Asynchronous Errors

The MXCC asserts the  $\overline{\text{AERR}}$  signal whenever an asynchronous error occurs. Asynchronous errors include errors of operation that the SSP has already acknowledged to the MXCC, but have errors occurring later in the operation. These errors are logged to the MXCC's error register (see Subsection 16.8.4) before they are reported to the system by asserting  $\overline{\text{AERR}}$ .

## Relinquish and Retry (R&R)

The MXCC supports R&R replies only on the first response to any transaction. Some caches, including the MXCC's E-cache, can present a problem when they receive an R&R acknowledgement. These caches no longer recognize the data block as owned when backed off a write-back of a cache block with an R&R. Other system logic must be responsible for ensuring that no other coherent transaction occurs to that block until the write succeeds. This problem arises due to the fact that the MXCC does not snoop the write-back buffer. The SSP in Mbus mode does not have this problem, since it snoops its store buffer.

## Retry

The MXCC does not support the retry reply. All retries are the equivalent of ERROR3 replies. The MXCC expects all data returned with a READY response to be truly correct. This data is used immediately (e.g., by storing in the E-cache or forwarding to the processor).

## 17.8 MBus Timing Summary

### **MBus Specification Rev 1.2**

Read	A+2 (min) for MRDY
Write	A+2 (min) - A+3 for different masters performing back to back writes for MRDY
Coherent Read	A+2 for MIF and/or MSF A+6 for MRDY A+2 for MRDY if MIF is not asserted
Coherent Read and Invalidate	A+2 for MIF No number specified for MRDY, Need to know when invalidation occurs.
Coherent Write and Invalidate	A+2 (min) - A+3 for different masters performing back to back writes for MRDY Need to know when invalidation occurs
Coherent Invalidate	A+2 for MRDY

### **SuperSPARC Master Directly on MBus**

*SuperSPARC asserts address, accepts an acknowledgement, accepts MIF, MSF, data*

Read	A+2 (min) for acknowledgement
Write	A+2 (min) for acknowledgement
Coherent Read	Cannot accept MIF after first MRDY (concurrent is acceptable) 4 cycles (min) after MIF for acknowledgement and data A+2 (min) for acknowledgement if no MIF
Coherent Read and Invalidate	Cannot accept MIF after first MRDY (concurrent is acceptable) 4 cycles (min) after MIF for acknowledgement and data
Coherent Write and Invalidate	Not issued by SuperSPARC due to write allocate cache scheme
Coherent Invalidate	A+2 (min) for acknowledgement if any is sent Note: SuperSPARC can accept at A+1, but systems normally cannot generate that fast.
acks	

### **SuperSPARC as a Snooping Cache Directly on MBus**

*SuperSPARC accepts address, asserts acknowledgement, asserts MIF, MSF, and sends data*

Read	Not valid
Write	Not valid
Coherent Read	A+3 for MIF and/or MSF (non-varying) A+7 for acknowledgement (non-varying)
Coherent Read and Invalidate	A+3 for MIF and/or MSF (non-varying) A+7 for acknowledgement (non-varying)

**Coherent Write and Invalidate**

Treated as Coherent Invalidate  
A+1 (SuperSPARC will not assert MRDY)

**Coherent Invalidate**

A+1 (No acknowledgement sent out)

**SuperSPARC Slave Directly on MBus**

*Read of Port Register*

Read A+3 (min) for acknowledgement

Write Not Valid

Coherent Read Not valid

Coherent Read and Invalidate  
Not Valid

Coherent Write and Invalidate  
Not Valid

Coherent Invalidate Not Valid

**MXCC Master on MBus**

*Module asserts address, accepts acknowledgement, MIF, MSF, and data*

Read A+2 (min) for any acknowledgement

Write A+2 (min) for any acknowledgement

Coherent Read  
Asynchronous Clock:  
A+2 (min) for MIF and/or MSF  
Cannot accept MIF after first acknowledgement (concurrent is acceptable)  
2 cycles (min) after MIF for acknowledgement and data  
Synchronous Clock:  
A+2 (min) for MIF and/or MSF  
Cannot accept MIF after first acknowledgement (concurrent is acceptable)  
2 cycles (min) after MIF for acknowledgement and data

Coherent Read and Invalidate  
Asynchronous Clock:  
A+2 (min) for MIF and/or MSF  
Cannot accept MIF after first acknowledgement (concurrent is acceptable)  
2 cycles (min) after MIF for acknowledgement and data  
Synchronous Clock:  
A+2 (min) for MIF and/or MSF  
Cannot accept MIF after first acknowledgement (concurrent is acceptable)  
2 cycles (min) after MIF for acknowledgement and data

Coherent Write and Invalidate  
A+2 (min) for acknowledgements  
Note: Block copy and block zero issue CWI

Coherent Invalidate A+2 (min) for acknowledgement

### **MXCC as a Snooping Cache on MBus**

*Module accepts address, asserts acknowledgement, MIF, MSF, and sends data*

Read	Not valid
Write	Not valid
Coherent Read	Asynchronous Clock: A+6 $\pm$ 1 for MIF and/or MSF A+14 for acknowledgement (best case) A+23 for acknowledgement (worst case) Synchronous Clock: A+6 for MIF and/or MSF A+16 for acknowledgement (best case) A+21 for acknowledgement (worst case)
Coherent Read and Invalidate	Asynchronous Clock: A+6 $\pm$ 1 for MIF and/or MSF A+14 for acknowledgement (best case) A+23 for acknowledgement (worst case) Synchronous Clock: A+6 for MIF and/or MSF A+16 for acknowledgement (best case) A+21 for acknowledgement (worst case)
Coherent Write and Invalidate	No acknowledgement sent out Note: Max of one CWI every forth cycle
Coherent Invalidate	No acknowledgement sent out Note: Max of one CI every forth cycle

### **MXCC Slave on MBus**

*Non-Coherent transactions. MXCC as addressed slave*

Read	A+9 to A+23 depending on register
Write	A+9 to A+23 depending on register
Coherent Read	Not valid
Coherent Read and Invalidate	Not valid
Coherent Write and Invalidate	Not valid
Coherent Invalidate	Not valid

# VBus

---

VBus is a non-multiplexed, synchronous, highly pipelined bus for a variety of system applications. VBus is an especially efficient connection between the SuperSPARC processor (SSP), external cache RAMs, and the MultiCache Controller (MXCC) in a SuperSPARC module.

Topic	Page
18.1 Introduction .....	18-2
18.2 VBus Signals .....	18-10
18.3 VBus Transactions and Waveforms .....	18-17



## 18.1 Introduction

VBus is a non-multiplexed, synchronous, highly pipelined bus that can be used for a variety of system applications. VBus has 36 pins for a physical byte address and 64 pins for data. VBus is especially tailored to provide an efficient connection between the SSP, the MXCC, and external cache in a SuperSPARC module.

VBus always operates synchronously with the SSP, and the processor clock provides all timing for the operations of VBus. A single clock generator provides a low-skew clock to all devices on a VBus. All signals on VBus are sampled on the rising edge of the VBus clock.

VBus is selected if **CCMODE** is high (deasserted state) when system reset (**RESET**) transitions from asserted (L) to deasserted (H). **CCMODE** should not be changed when **RESET** is high (deasserted state), as unpredictable operation may result. The selection of MBus or VBus is visible to software in the **MCNTL** register as the **mb** bit.

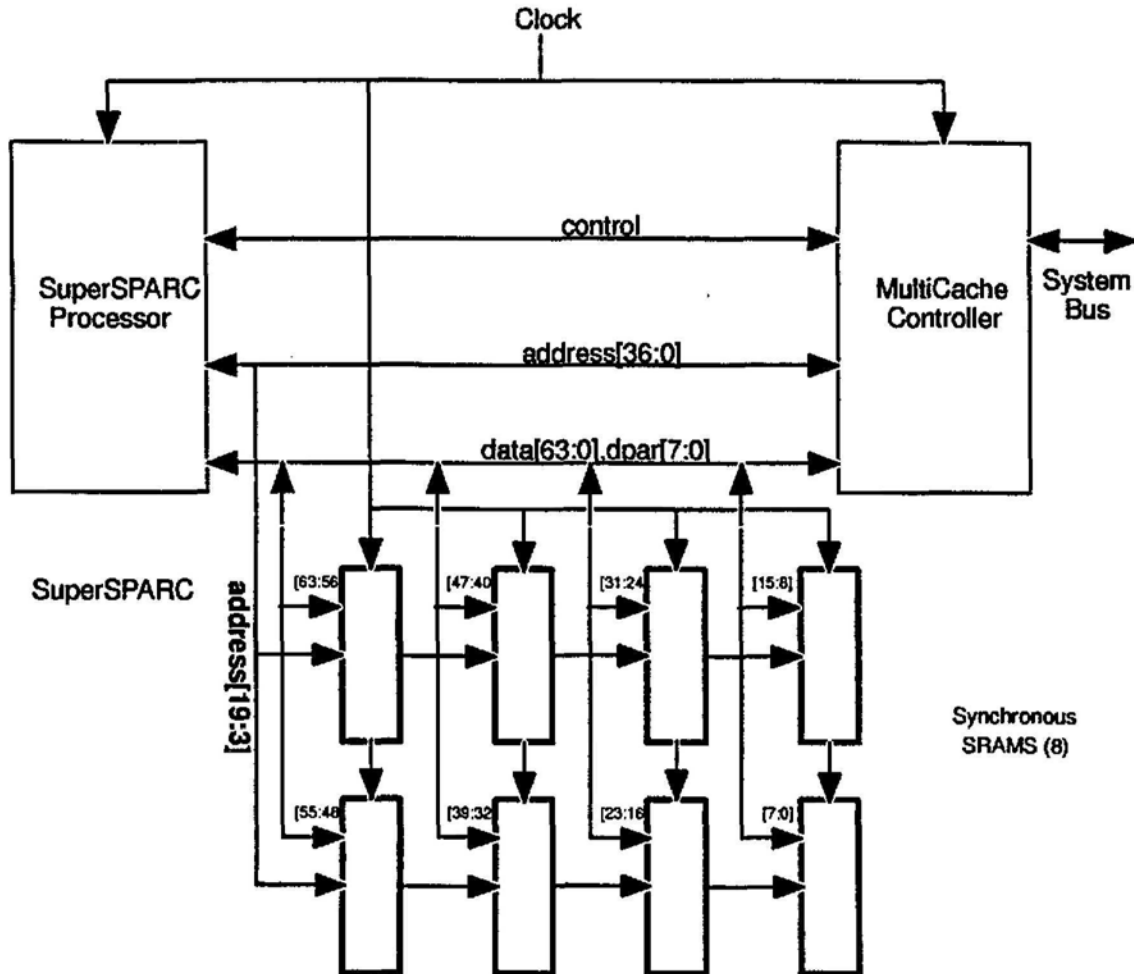
A central VBus arbiter controls access to the bus. A master must have **RGRT** (read grant) to begin a read access or **WGRT** (write grant) to begin a write access. The MXCC contains an integrated VBus arbiter.

The following sections introduce the basic transactions on VBus. The interaction of these transactions with the cache-consistency protocol will also be described.

### 18.1.1 Synchronous SRAM External Cache

In order to understand the operation of VBus, it is helpful to understand the function and the organization of the SSP, the MXCC, and SRAMs on the VBus. Generally, the SSP is a VBus master that accesses system memory and other resources through the MXCC. The bussed address and data lines go to the external cache SRAMs and the MXCC. Figure 18-1 shows the general scheme of VBus connections.

Figure 18-1. VBus Sub-system

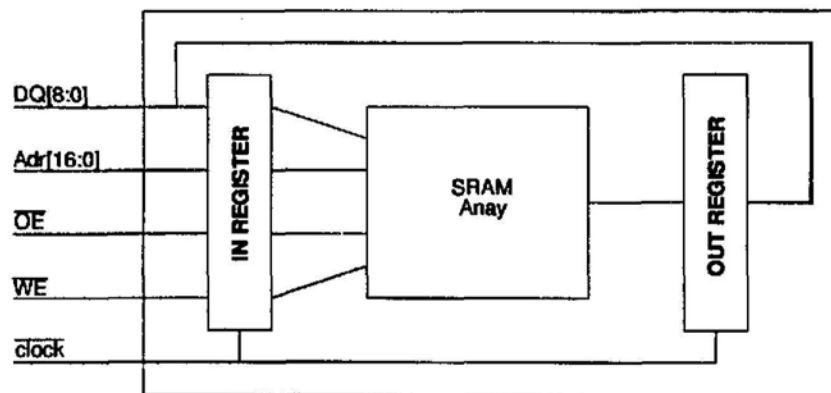


When a cache hit occurs, the MXCC times the local Static Random Access Memory (SRAM) accesses by providing ready responses and write-enable gating. The MXCC also functions to control cache fills and assumes the responsibility to write data into SRAMs when the SSP issues write to shared data. The MXCC controls the processor's access to the VBus with the grant lines.

A 1MB E-cache consists of eight 128Kx8 or 128Kx9 synchronous SRAMs, as shown in Figure 18-1. Each of the SRAMs DQ[7:0] pins connects to one of the D[63:00] pins. The ninth bit of the 128Kx9 SRAMs should be connected to the corresponding DPAR signal. The SRAM's output enables are controlled by the OE signal from either the MXCC or the SSP, and their write enables are controlled by WE [7:0] from either the MXCC or the SSP.

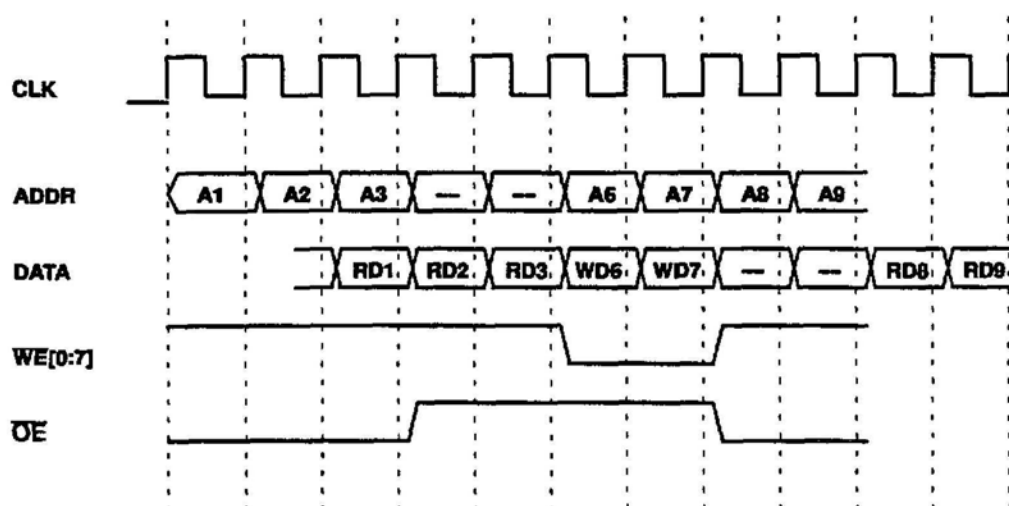
Synchronous SRAMs have registers on each input and output, as shown in Figure 18-2. This allows pipelined operation. An address is presented to an SRAM before the active clock edge, and it is registered in the input register at the clock edge; the result is stored in an output register at next active edge. New addresses can be supplied at each clock edge, and new outputs appear after two clock periods of delay. Writing works similarly, with address, data, output enable, and write enable being registered on the active clock edge and stored into the internal array during the subsequent clock period.

Figure 18-2. Synchronous SRAM Internal Organization



The timing of synchronous SRAM operation is illustrated in Figure 18-3. It shows three read cycles, followed by two write cycles, followed by two read cycles, all proceeding as quickly as possible. Notice that, when changing from reading to writing, two address cycles must pass with  $\overline{OE}$  low before the write address and data can be placed on the bus. These two cycles are needed for the data from the last read to appear on the data bus. Notice also that the data bus, when changing from writing to reading, is idle for two cycles because it takes two cycles before the data from the first read is available on the bus.

Figure 18-3. Synchronous SRAM Timing



Synchronous SRAMs having this organization and sufficient speed to operate at the VBus clock frequency may be used with the SSP and MXCC. There are several manufacturers of suitable components.

The timing of the VBus when used to access the synchronous SRAMs is not identical to that shown in Figure 18-3, since the MXCC controls access to the VBus with  $WGRT$  and  $RGRT$  and controls the access to the data bus with  $WEE$ . The actual timing on VBus of a write following a read is shown in Figure 18-28.

### 18.1.2 Bus Transactions

Read and write single transactions are the most basic bus operations. Block read and burst write transactions are used to improve VBus bandwidth. Most transactions are pipelined to provide higher performance. A swap (atomic load/store) operation is defined to provide locked bus transactions.

A transaction on VBus begins with a command cycle. In a command cycle, the address is supplied along with the command status lines. The data lines may also be driven for write commands. A command cycle is indicated by the **CMDS** signal. The slave decodes the address and command status signals and responds with an acknowledgement only for write transactions or data and an acknowledgement for read transactions.

Burst transactions involve more than one doubleword (DW) and are of variable size. In a burst transaction, two or more DWs of data are transferred after a single command cycle. The burst continues for one more DW if **BURST** remains asserted or is complete after the next acknowledgement if **BURST** is deasserted.

### 18.1.3 Cache Consistency

When using the VBus, the processor's internal data cache operates as a write-through cache. Since the cache writes through, all modifications that the processor makes to cached data appear externally as VBus write transactions. These transactions are used to keep all caches (internal caches, external caches, and the caches of other processors) consistent with the new changes.

The SuperSPARC processor's internal caches are kept consistent by invalidation. Whenever an external bus master asserts the **CMDS** and **WR** signals (with **DEMAP** deasserted), any copies of data cached internally that match the address on the bus will be invalidated.

In systems with external caches, inclusion is normally maintained with the external cache. This allows the external cache controller to filter many system bus transactions and pass on only those that require action within the first-level caches to the SuperSPARC processor. The MXCC works in this way.

### 18.1.4 Error Reporting

Each VBus transfer is acknowledged with a response. The response indicates good completion or one of several types of error. Each transaction has one or more responses, depending on the number of DWs transferred.

The transfer responses are encoded on the **RRDY**, **WRDY**, **RETRY**, and **MEXC** signals. This encoding is similar to the encoding defined for MBus error responses. **RRDY** and **WRDY** are interpreted in the same way, but **RRDY** applies to read transactions and **WRDY** applies to write transactions. Table 18-1 defines the encoding of VBus acknowledgements and errors.

*Table 18-1. Bus Transfer Responses*

<b>MEXC</b>	<b>WRDY/RRDY</b>	<b>RETRY</b>	<b>REPLY DEFINITION</b>
H	H	H	No Reply
H	H	L	Retry
H	L	H	Data Transfer complete
H	L	L	UD (Undefined) Error
L	H	H	BE (Bus Error)
L	H	L	TO (Timeout) Error
L	L	H	UC (Uncorrectable) Error
L	L	L	Reserved

The undefined (UD), bus error (BE), timeout (TO), and uncorrectable[?] (UC) errors are not actually interpreted by the logic in the SSP. Each is logged in the appropriate bit in the fault status register (MFSR) but generates no other specific actions that differentiate the four errors. Thus, the names given each of the four errors are only suggestions, and systems may choose other interpretations of the four codes.

Throughout the bus cycle examples in this chapter, a standard error response is used when the example shows an error response. The error shown has **MEXC** and either (or both) **RRDY** or **WRDY** asserted, which is the encoding for an uncorrectable error. Other than retry responses, the error replies are all treated in exactly the same manner. The particular error type is decoded and used to set the appropriate bits in the MFSR.

### 18.1.5 Memory and I/O Transactions

Memory transactions are generally cacheable; I/O transactions generally are not. Cacheable read operations, with the exception of Memory Management Unit (MMU) table walk operations, use read/block transactions. Non-Cacheable transactions always use single-cycle transfers. The **CCHBL** signal is asserted during all cacheable transactions.

### 18.1.6 Arbitration

When the SSP needs to perform an external bus access, it must have access to the system bus. Access is granted to the processor by asserting one or both of the bus grant signals **RGRT** and **WGRT**. In systems with external caches, these signals will normally be controlled independently. In simpler Dynamic Random Access Memory (DRAM)-based systems, a single composite **BUSGRT** signal may be used by connecting **WGRT** and **RGRT** together.

The SSP supports lazy arbitration; if the bus grant is already asserted when the SSP wants to begin an access, the access will begin immediately. If a grant is not present, the **BUSREQ** signal will be asserted until the bus is granted. Once the bus is granted, the SSP may issue an access, as described in the following sections. The bus grant must remain active for the duration of the bus cycle. If the grant signal is deasserted, the SSP disables its drivers on all shared VBus signals on the following cycle.

---

**Note:**

The SuperSPARC processor begins a transaction on VBus immediately if the bus grant signal needed is already active. Since it is possible for SuperSPARC to begin a bus cycle at the same time external arbitration is removing these bus grants, system logic must monitor the **CMDS** signal to ensure that a transaction had not started as the bus grant was removed. Since external arbiters should ensure that an empty cycle exists between bus owners, this cycle may be used to detect the arbitration collision.

Once the bus grant has been deasserted, the SSP will disable its I/O signals on the next cycle. This could interfere with the transaction that was started, although some part of the transaction may have completed. When this situation occurs, you should retry the SSP's transaction. When the memory or external cache controller detects the arbitration conflict (grant is deasserted and **CMDS** is asserted), it should immediately assert the **RETRY** signal. This will force SuperSPARC to cancel the pending transaction and re-arbitrate for use of the bus. See Figure 18-9.

---

The SSP also samples the **OE** pin to prevent collisions with returning read data from reads generated by the external cache controller. Although it is generally legal for the SSP to overlap write cycles with outstanding read miss requests, this can cause a data drive conflict when the external cache controller is reading from the SRAM and the SSP tries to begin a store transaction. The SSP does not initiate driving the data bus if the **OE** signal was asserted externally on the previous cycle. This case should generally be covered by arbitration but is included for extra protection.

### **18.1.7 MMU Consistency**

Demap transactions are used by the SSP to flush one or more pages from its MMU's translation lookaside buffer (TLB). Internally, these are generated by reference MMU flush operations (see Section 8.8). While processor demap operations do not perform bus transactions in MBus systems, in a VBus configuration internal demap operations also generate VBus demap transactions. The purpose of a VBus demap transaction is to communicate demap operations from the SSP to other TLBs in the system or to communicate demap operations originating elsewhere in the system to the local SSP.



## 18.2 VBus Signals

The SSP's interface to the system when VBus is selected is 143 signals plus the processor (and VBus) clock VCLK. These signals use CMOS levels. All signals are fully synchronous and are sampled on the rising edge of the VBus clock, VCLK.

Most VBus bidirectional signals have bus keepers which are low power amplifiers that keep the voltage level signal pins from drifting from valid high or low values when all drivers are in a high-impedance state.

<b>ADDR[35:0]</b>	Address. These 36 signals provide the physical byte addresses for all VBus transactions. They are sampled when <b>CMDS</b> is asserted and on successive cycles if <b>BURST</b> is asserted. These signals are I/Os on both the SSP and the MXCC.
<b>DATA[63:0]</b>	Data. These are the 64 data bits for VBus transactions. Byte-wise parity is computed and appears on <b>DPAR[7:0]</b> . These signals are I/Os on both the SSP and the MXCC.
<b>ARDY</b>	This signal is an input to the SSP. When used with the MXCC, <b>ARDY</b> should be tied to low, as the MXCC is always ready to accept addresses or bus cycles. The MXCC has no <b>ARDY</b> pin. It is recommended that this signal always be tied low.
<b>BURST</b>	This signal is an output from the SSP and an input to the MXCC. It indicates that the current address on the bus is part of a burst bus cycle. <b>BURST</b> is asserted the same time as <b>ADDR[35:0]</b> ; it is asserted through both read and write bursts. <b>BURST</b> is deasserted on the last address of a burst to allow the MXCC to stop returning <b>RRDY</b> or <b>WRDY</b> with the last data of the burst.
<b>CCACHE</b>	This signal is an output from the SSP and an input to the MXCC. It indicates that the current transaction is cacheable in an external cache. This signal is sampled by MXCC only when <b>CMDS</b> is asserted by the SSP.

**CMDS**

Command Strobe. The VBus master asserts this signal for one cycle to begin all transactions. If the SSP is not the bus master, the MXCC asserts CMDS as an input to the SSP to initiate external snoop transactions (including both demaps and invalidates). With the MXCC as VBus master, CMDS high indicates that no command word is present on the VBus. CMDS low indicates an MXCC-initiated VBus invalidate or demap command word on ADDR[35:0], DEMAP, and WR.

When the SSP is the bus master, it asserts this signal for the first cycle of a VBus transaction. In this case, CMDS high indicates that no command word is present on the VBus. CMDS low indicates a SuperSPARC initiated VBus command word on ADDR[35:0], CCHBL, CSA, DEMAP, LDST, SIZE[1:0], SU, RD, and WR.

**CSA**

Control Space Access. the SSP asserts this signal when performing a read or write to the MXCC control space through ASI 0x02. The E-cache tag RAM, E-cache data, and internal MXCC registers are accessible through ASI 0x02. When CSA is high, a normal memory access is indicated, whereas when CSA is low a control space access is indicated.

**DEMAP**

Demap an address translation. DEMAP is asserted along with CMDS to indicate a demap cycle. This cycle can be initiated or by the system (as communicated via the MXCC) or the local SSP. The DATA[63:0] lines contain a demap data word; the demap data word indicates which virtual address translations are to be discarded.

When DEMAP is input to the SSP (output from the MXCC), any entries in the SSP's TLB matching the request will be removed.

When the SSP initiates a **DEMAP** transaction, translation hardware in the system should remove TLB entries matching the request. When **DEMAP** is input into the MXCC and **WR** asserted, a demap request is passed from SuperSPARC to the MXCC and out to the system bus. A **DEMAP** assertion input into the MXCC, with **RD** asserted, indicates that SuperSPARC has successfully completed a demap operation initiated by the system bus (through the MXCC).

**DEMAP** is sampled only when **CMDS** is asserted.

**DPAR[7:0]**

Data bus parity. When parity is enabled, even parity is generated and checked. The correspondence of DPAR bits to the DATA bits checked are shown in Table 18-2.

Table 18-2. DATA Bits Checked by DPAR Bits

Parity Bit	Checks Data Bits
DPAR[0]	DATA[63:56]
DPAR[1]	DATA[55:48]
DPAR[2]	DATA[47:40]
DPAR[3]	DATA[39:32]
DPAR[4]	DATA[31:24]
DPAR[5]	DATA[23:16]
DPAR[6]	DATA[15:8]
DPAR[7]	DATA[7:0]

**ERROR**

Processor Error. This signal is asserted by the SSP to indicate that the processor has entered error mode and will take a watchdog reset trap. The MXCC initiates an internal error when **ERROR** is asserted. See Section 12.3.

**IRL[3:0]**

Interrupt Request Level. This field specifies the level of the highest priority interrupt request that is currently pending. It is an input to the SSP. If **IRL[3:0] = 0000**, no interrupts are pending. Level 15 (**IRL[3:0] = 1111**) indicates a non-maskable interrupt (NMI) that cannot be masked by **PSR.PIL**; level 14 is the highest priority maskable interrupt; level 1 is the lowest priority maskable interrupt.

**LDST**

This signal indicates that an atomic Load/Store operation (LDSTUB, LDSTUBA, SWAP, or SWAPA) has been initiated by the SSP. LDST is the equivalent of a logical OR of the RD and WR signals.

**MEXC**

Memory Exception. This signal is output from the MXCC to the SSP and is asserted when the MXCC cannot return or accept the requested data. This signal may cause the SSP to take an exception trap. MEXC is encoded, along with RRDY or WRDY, and RETRY to indicate the type of acknowledgement for a transaction initiated by the SSP (See Table 18-3).

Table 18-3. VBus Acknowledgements

MEXC	RRDY/WRDY	RETRY	Description
H	H	H	No Reply
H	H	L	Retry
H	L	H	Data Transfer Complete
H	L	L	UD (Undefined) Error
L	H	H	BE (Bus Error)
L	H	L	TO (Timeout) Error
L	L	H	UC (Uncorrectable) Error
L	L	L	Reserved

**OE**

SRAM Output Enable. As an output from either the SuperSPARC processor or the MXCC, this signal controls the pipelined output enable of the external cache SRAMs. OE is used as an input by the SSP to prevent bus collisions.

**PEND**

Pending. This signal is output from the MXCC to inform the SSP that there is at least one outstanding write operation that has not completed. PEND is asserted by the MXCC when it has a store operation pending internally or on the system bus. PEND is used by the SSP to support the PSO memory model (see Section 7.7).

<b>RD</b>	Read. The RD signal is driven to qualify addresses on the VBus as read cycles. It is also asserted by the SSP, along with WR and LDST, to indicate an atomic Load/Store cycle. The SSP uses RD as an input for factory SRAM test only. RD is sampled only when CMDS is asserted.
<b>RESET</b>	Reset. This input forces the SSP to perform hardware reset. See Chapter 12.
<b>RETRY</b>	Retry. This signal is encoded, along with RRDY/WRDY and MEXC, to indicate the type of acknowledgement on VBus. See Table 18-3. If the MXCC asserts this signal before RRDY or WRDY is asserted for an access, the processor terminates the current access and restarts it once it reacquires the VBus (if a read is pending, a write will not be retried until after the read has completed).
<b>RGRT</b>	Read Grant. This signal grants read access on VBus to the SSP.
<b>RRDY</b>	Read Ready. This signal indicates to the SSP that read data is valid. When RRDY is asserted, the SSP may reliably sample the incoming data on the same clock edge as RRDY. This signal is used to qualify data specifically for a read access, since a write may also be pending. This signal is encoded with MEXC and RETRY to provide the MXCC transaction acknowledgements, as in Table 18-3.
<b>SIZE[1:0]</b>	These bits indicate to the MXCC the size of the current VBus transaction initiated by the SSP. They are outputs only. This field is only sampled when CMDS is asserted. The encoding of the SIZE field is shown in Table 18-4.

Table 18-4. Size Encodings

SIZE[1:0]	Transaction Size
00	byte (8-bit)
01	halfword (16-bit)
10	word (32-bit)
11	doubleword (64-bit)

<b>SU</b>	Supervisor Access. This signal is asserted by SuperSPARC with <b>CMD<sub>S</sub></b> when the access was initiated with <b>PSR.S=1</b> (in supervisor mode).
<b>WE[7:0]</b>	SRAM Write Enables. These signals directly control the write-enable signals of the synchronous SRAM used for the external cache. These signals are only driven when asserted; otherwise, they are in a high-impedance state. <b>WE</b> bit ordering conforms to the big-endian convention ( <b>WE0</b> is the write enable for byte 0 – <b>DATA[63:56]</b> )
<b>WEE</b>	E-cache write enable enable. The SSP cannot place write data onto <b>DATA[63:0]</b> and <b>DPAR[7:0]</b> or drive the <b>WE[7:0]</b> pins until permitted by the MXCC using <b>WEE</b> . During a write transaction on VBus, the SSP will wait for <b>WEE</b> to be asserted before using those signals on VBus. Once <b>WEE</b> has been asserted, the SSP has permission to use VBus for writing for the duration of the burst if needed. See Subsection 18.3.7 for figures illustrating the use of <b>WEE</b> .
<b>WGRT</b>	Write Grant. This signal is output by the MXCC and allows the SSP to begin a write access on the VBus.
<b>WR</b>	Write. The SSP drives the <b>WR</b> signal to qualify an address on VBus as a write cycle. When driven by the SSP, along with <b>DEMAP</b> , a demap request is sent to the system bus. When <b>WR</b> is driven by the SSP, along with <b>RD</b> and <b>LDST</b> , an atomic Load/Store transaction is sent. The MXCC will output <b>WR</b> , with an address on <b>ADDR[35:0]</b> , to invalidate lines in SuperSPARC's internal caches that contain that address.
<b>WRDY</b>	Write Ready. This signal notifies the SSP that the MXCC has sampled the SSP's write data, and the processor may generate the next access. In the case of burst writes, the processor will switch address and data for the next write within the burst on the same clock edge on which <b>WRDY</b> was asserted. This signal is used to qualify data specifically for a write access, since a read may also be pending. This signal is encoded with <b>MEXC</b> and <b>RETRY</b> to provide the MXCC transaction acknowledgements, as in Table 18-3.

Table 18–5 summarizes the VBus signals and which devices use the signal as an input, output, or I/O. Entries labelled O/Z are outputs that are high-impedance except when asserting the signal or during the 1/2 cycle restoration of the deasserted level on the bus.

■ Table 18–5. Summary of VBus Signals

Signal	SSP	MXCC	SRAMs
ADDR[35:0]	I/O	I/O	I
DATA[63:0]	I/O	I/O	I/O
ARDY	I	—	—
BURST	O	I	—
CCRBL	O	I	—
CMDS	I/O	I/O	—
CSA	O	I	—
DEMAP	I/O	I/O	—
DPAR[7:0]	I/O	I/O	I/O
ERROR	O	I	—
IRL[3:0]	I	O	—
LDST	O	I	—
MEXC	I	O	—
OE	I/O	O/Z	I
PEND	I	O	—
RD	I/O	I	—
RESET	I	O	—
RETRY	I	O	—
RGRT	I	O	—
RRDY	I	O	—
SIZE[1:0]	O	I	—
SU	O	I	—
WE[7:0]	O/Z	O/Z	I
WEE	I	O	—
WGRT	I	O	—
WR	I/O	I/O	—
WRDY	I	O	—

## 18.3 VBus Transactions and Waveforms

In order to understand the following waveforms, it is helpful to understand the function and the organization of the SSP, the MXCC, and SRAMs on the VBus. Generally, the SSP is a VBus master that accesses system memory and other resources through the MXCC. The bussed address and data lines go to the external cache SRAMs and the MXCC.

When a cache hit occurs, the MXCC functions to time the local SRAM accesses by providing ready responses and write-enable gating ( $\overline{WEE}$ ). The MXCC also functions to control cache fills and assumes the responsibility to write data into SRAMs when the SSP issues write to shared data. The MXCC controls the processor's access to the VBus with the grant lines  $\overline{RGRT}$  and  $\overline{WGRT}$ . The MXCC will deassert the grants when it needs access to the SRAMs (either read or write), when it needs to issue system DEMAP requests or system DEMAP replies, or when it needs to invalidate entries in the SSP's internal caches. The MXCC VBus cycles are not normal master/slave bus cycles because there is no acknowledgement to the MXCC from any slave device on the VBus.

A 1 Mbyte E-cache consists of eight 128Kx8 or 128Kx9 synchronous SRAMs. Each of the SRAM's DQ[7:0] pins connects to one of the D[63:00] pins. The ninth bit of the 128Kx9 SRAMs should be connected to the corresponding DPAR[7:0] signal. The SRAMs' output enables are controlled by the  $\overline{OE}$  signal from either the MXCC or the SuperSPARC processor; their write enables are controlled by  $\overline{WE}$  [7:0] from either the MXCC or the SSP.

### 18.3.1 VBus Waveforms

All signals on the VBus are sampled on the rising edge of the internal VBus clock signal (VCLK on the SSP and PCLK on the MXCC). When  $\overline{PLLBYF}$  is asserted, the internal clock is a delayed (by about 4.5 ns) version of the VBus clock as presented to the VCLK or PCLK pin. When  $\overline{PLLBYF}$  is deasserted, the internal clock is generated by the phase-locked loop to be coincident with the rising edge of the PCLK pin when  $\overline{PLLBYF}$  is not asserted. See Chapter 21.

The MXCC controls access to VBus through the  $\overline{RGRT}$  and  $\overline{WGRT}$  lines. SuperSPARC may access VBus for a read when  $\overline{RGRT}$  is asserted and may access the bus for a write when  $\overline{WGRT}$  is asserted.



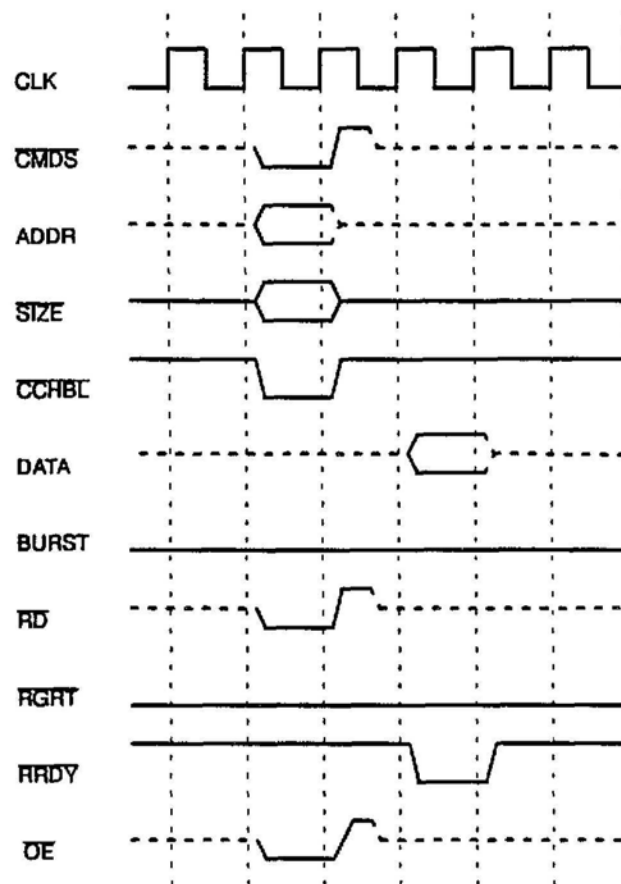
VBus transactions begin when **CMD $\overline{\text{S}}$**  is asserted and **ADDR[35:0]** and **SIZE[1:0]** are driven. Other signals (**RD**, **WR**, **CCHBL**, **BURST**, **LDST**, **CSA**, **DPAR[0:7]**, and **DATA[63:00]**) are also driven as required for this type of cycle. Since bus keepers maintain undriven signals at a valid level (either high or low), the SSP will only drive signals that need to be asserted. Shared signals that are pulsed (for example, **CMD $\overline{\text{S}}$** ) are driven to the deasserted level for one half cycle before being released to undriven. During writes, **ADDR[35:0]**, **SIZE[1:0]**, **D[63:0]**, **RD**, **WR**, **CCHBL**, and **DPAR[7:0]** are held valid until acknowledged by **WRDY**.

In the following waveforms, the **ADDR**, **DATA**, and **WE** buses are shown as single signals to simplify the diagrams.

### 18.3.2 Cacheable Single Read Hit

Figure 18-4 shows a read by the SSP of a single cacheable word with an E-cache hit. The processor asserts the address, cycle qualifiers, and the  $\overline{OE}$  to SRAM. The MXCC detects an E-cache tag match and issues a  $\overline{RRDY}$  at the same time that the SRAMs drive data to SuperSPARC. The  $\overline{OE}$  from the SSP is delayed in the registers internal to the synchronous SRAMs, and the data is enabled two cycles after the  $\overline{OE}$  is issued to the SRAMs. Note that the partially bussed VBus control signals are actively deasserted for 1/2 cycle before being released to the bus keepers.

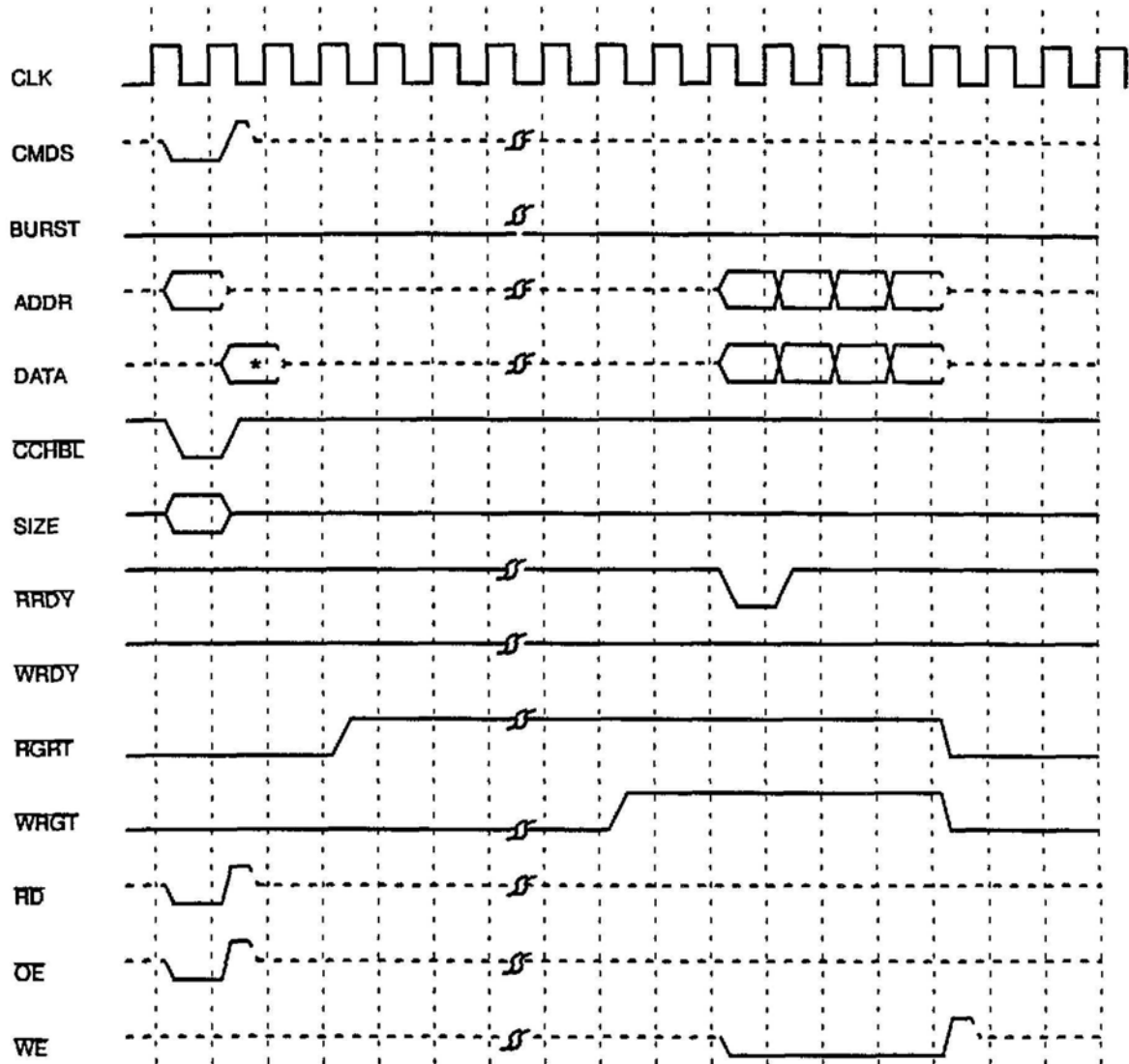
Figure 18-4. VBus-Cacheable Single Read Hit



### **18.3.3 Cacheable Single Read Miss**

Figure 18-5 shows a cacheable single-read miss. The MXCC detects that a tag mismatch occurs and issues a cycle to the system bus to obtain data to fill the E-cache. It removes  $\overline{\text{RGRT}}$  to allow SuperSPARC to proceed with any write operation it may have had pending. When the system bus returns the requested data block, the MXCC removes bus grant to SuperSPARC (negates  $\overline{\text{WRGT}}$ ) to obtain access to the SRAMs. The MXCC writes the data into the SRAMs. The MXCC issues a  $\overline{\text{RRDY}}$  to SuperSPARC, as the data word requested (by SuperSPARC read) is driven on the DATA lines (while the data is being written into SRAMs).

Figure 18-5. VBus-Cacheable Single Read Miss

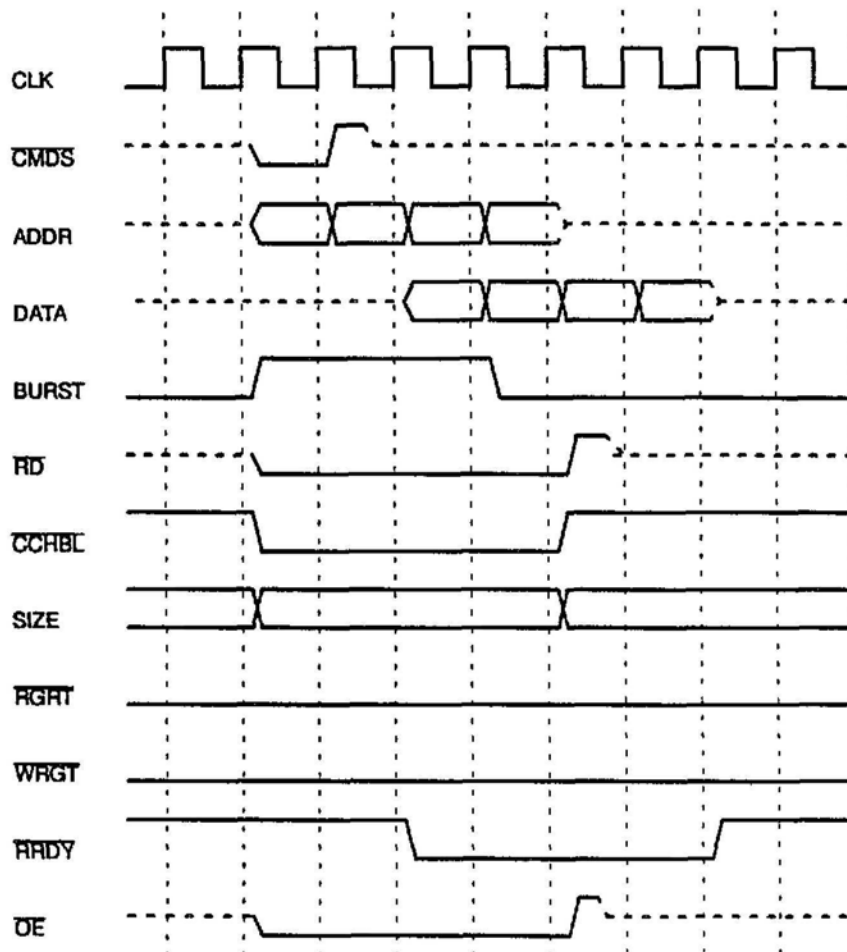


\* Incorrect data, ignored

### 18.3.4 Burst Read Hit

Figure 18–6 shows a burst read hit. As with a cacheable single read hit, the MXCC functions mainly to time the cycle by asserting **RRDY** as the SRAM provides the data.

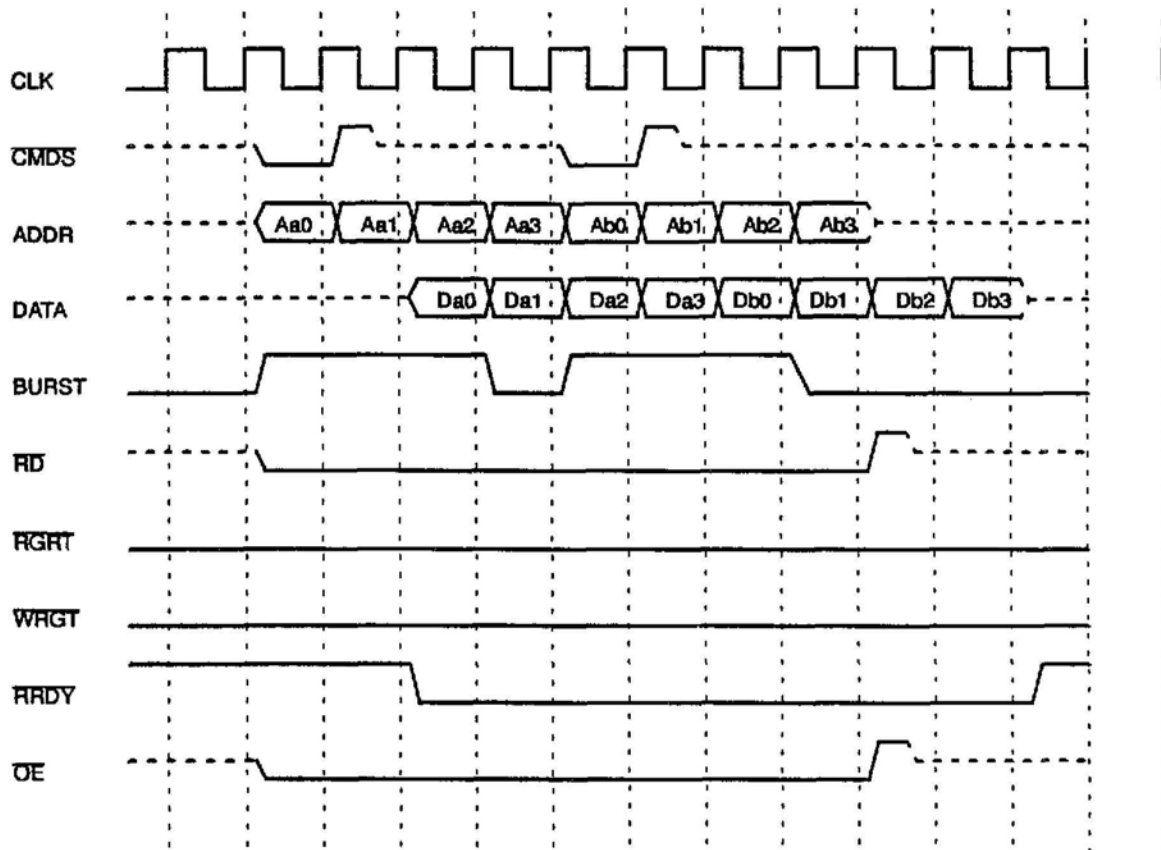
Figure 18–6. VBus Burst Read Hit



**Overlapped Burst Read Hit**

Figure 18-7 shows a second burst read cycle overlapping the first. In this example SuperSPARC issues a **CMDS** and next burst address as soon as the last of the previous addresses was sent. The earliest the overlapping cycle can occur is one cycle after the first ready (**WRDY** or **RRDY**) has been received by SuperSPARC. In Figure 18-7 the overlap is regulated by the availability of the address lines.

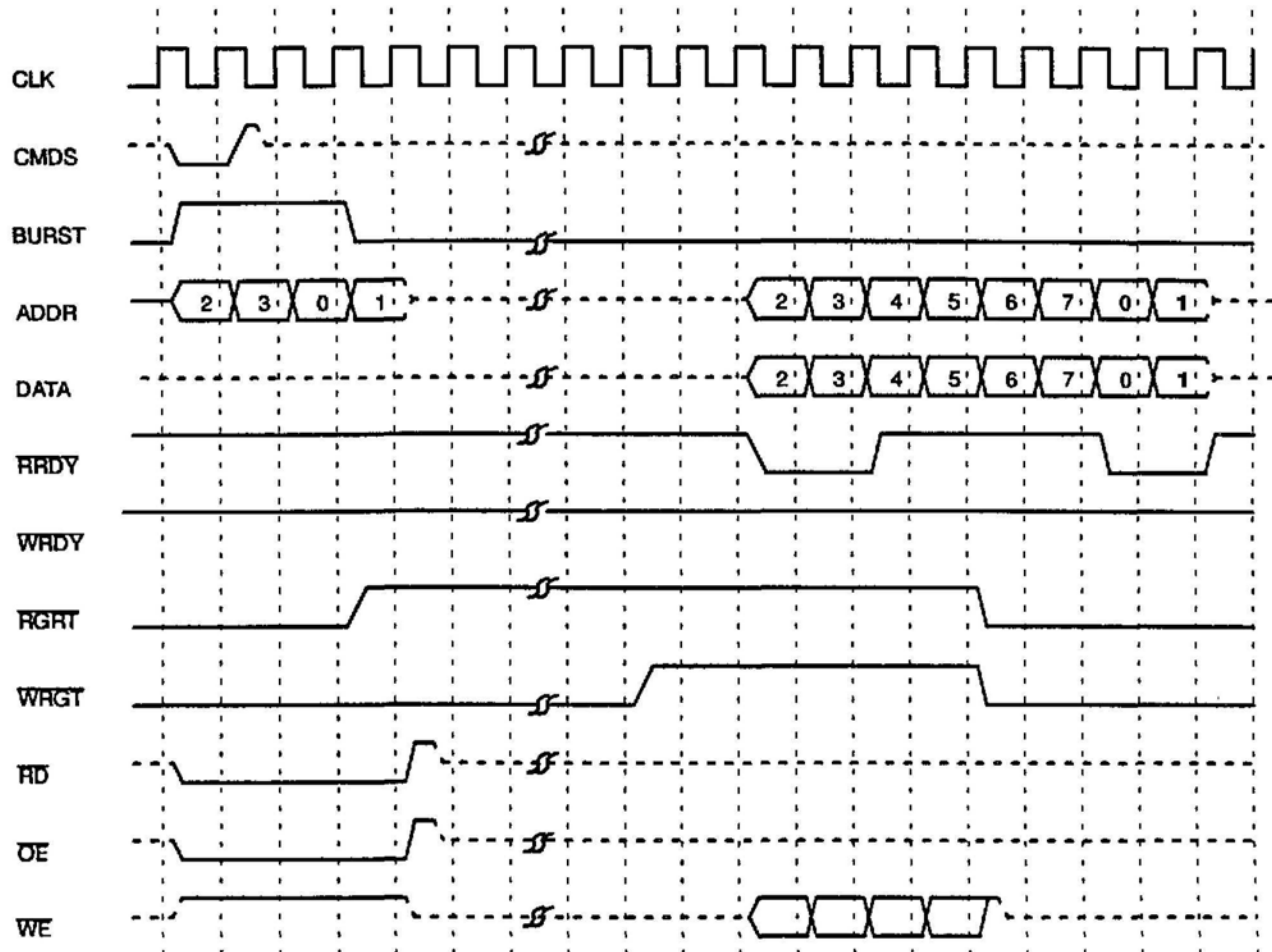
Figure 18-7. VBus-Overlapped Burst Read Hit



### **18.3.5 Burst Read Miss**

Figure 18–8 shows a burst read miss. The MXCC removes **RRGT** to indicate that the cycle is in progress and that SuperSPARC can proceed with an outstanding write if one is pending. When the data returns from the system bus, the MXCC writes it into the SRAM and asserts **RRDY** when the requested data is on the VBus. Note that, in Figure 18–8, the MXCC is in XBus configuration, and consequently the block size is 64 bytes. Only 32 bytes are sent to SuperSPARC, while all 64 bytes are stored in SRAM. Also note that with critical word first ordering that the data returned starts from the index into the block for the requested doubleword, continues to the last index, and then wraps from index 0 to the starting index minus 1.

Figure 18-8. VBus Burst Read Miss

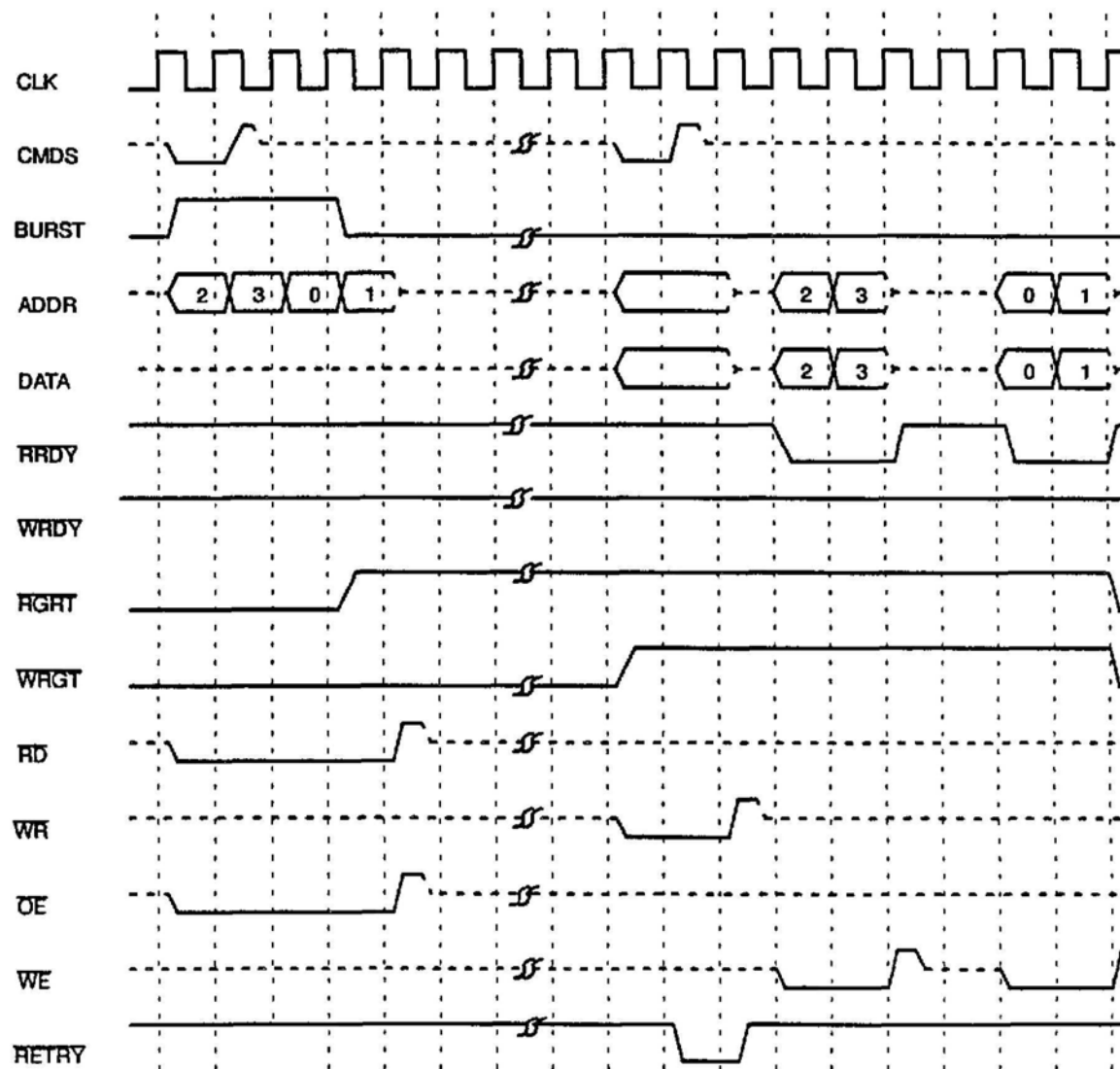


P Read is critical word first ordering. This example has word two of the block returned first.



Figure 18-9 shows a burst read miss that detects a VBus operation at the same time it is removing write grant. The MXCC must issue a retry and wait an extra clock to allow the SSP to get off the bus. Figure 18-9 depicts an MBus configuration (i.e., 32 bytes of data are returned from the system bus). The interruption in the transaction can occur due to the difference in clock frequencies between the VBus and MBus.

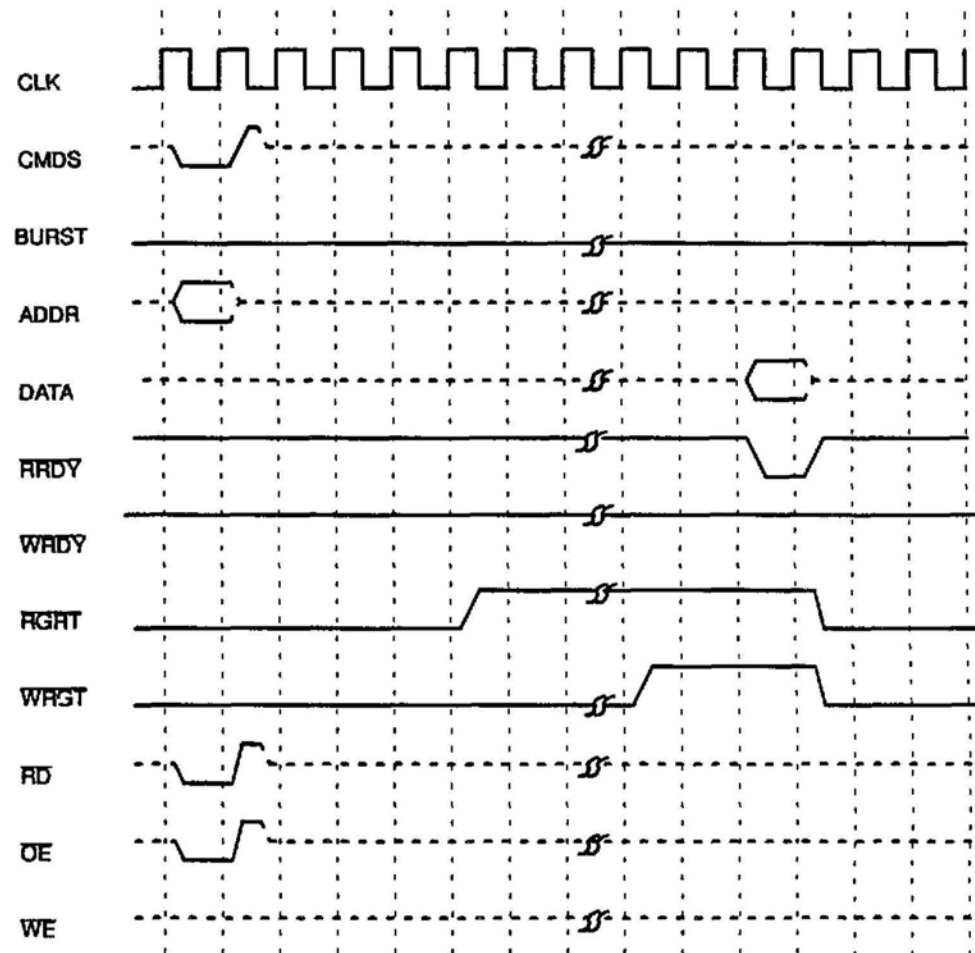
Figure 18-9. VBus Read Miss (With interference from the SuperSPARC processor)



### 18.3.6 Cache Disable Read

Figure 18-10 shows a single read with the cache disabled. The MXCC goes to the system bus to accomplish this operation. It deasserts **RGRT** to allow SuperSPARC to complete pending write operations. When the data is available, the MXCC negates grant, drives the data, and asserts **RRDY**.

Figure 18-10. VBus Cache Disable Read



### 18.3.7 Cacheable Write Hit

Figure 18–11 shows a cacheable single-write hit. The MXCC asserts  $\overline{WEE}$  at the CMD+2 cycle (i.e., two cycles after CMD $\overline{S}$ ) to allow the assertion of the write data (DATA, DPAR) and strobes ( $\overline{WE}[0:7]$ ). The MXCC asserts the  $\overline{WRDY}$  in the following cycle (CMD $\overline{S}$  + 3).

Figure 18–11. VBus-Cacheable Single Write Hit

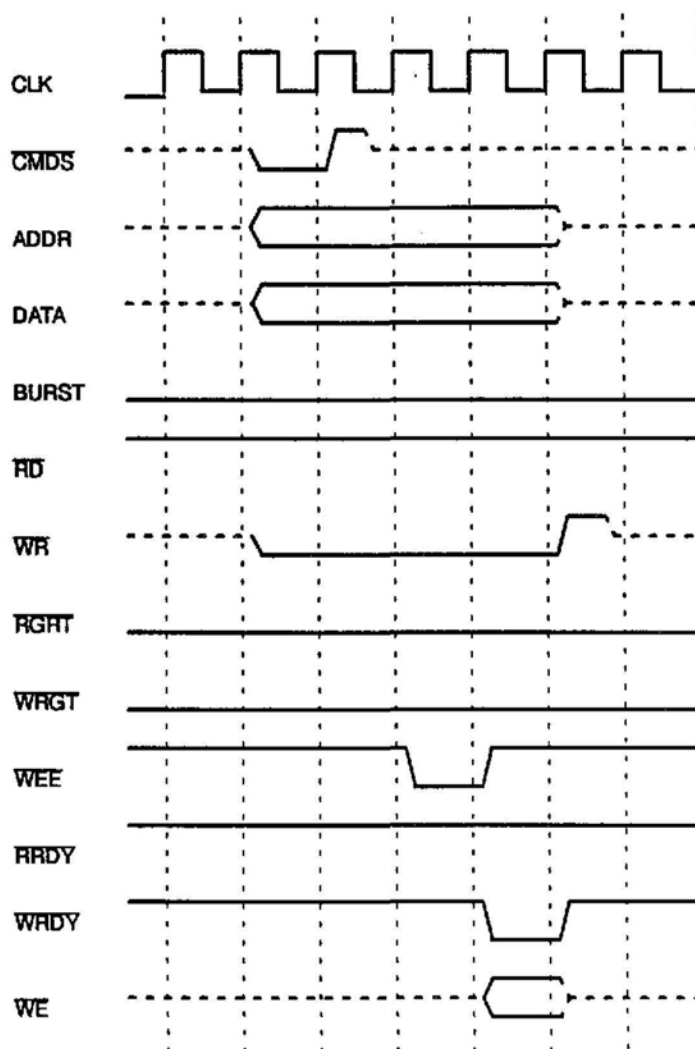


Figure 18-12 shows a burst hit. It is basically the same except that **WRDY** is asserted for each data doubleword written in the burst. The SSP deasserts **BURST** one cycle before the last write. Each of the individual writes in the burst from the SSP may be from one to eight bytes and may be at any address within the cache block. The number of consecutive writes may be of arbitrary length. If the MXCC needs the VBus while a burst write cycle is occurring, it can deassert the **WRGT** signal to terminate the burst cycle prematurely, as shown in Figure 18-13. When the SSP reacquires the VBus, it continues the burst write from where it was interrupted.

Figure 18–12. VBus-Cacheable Write Hit

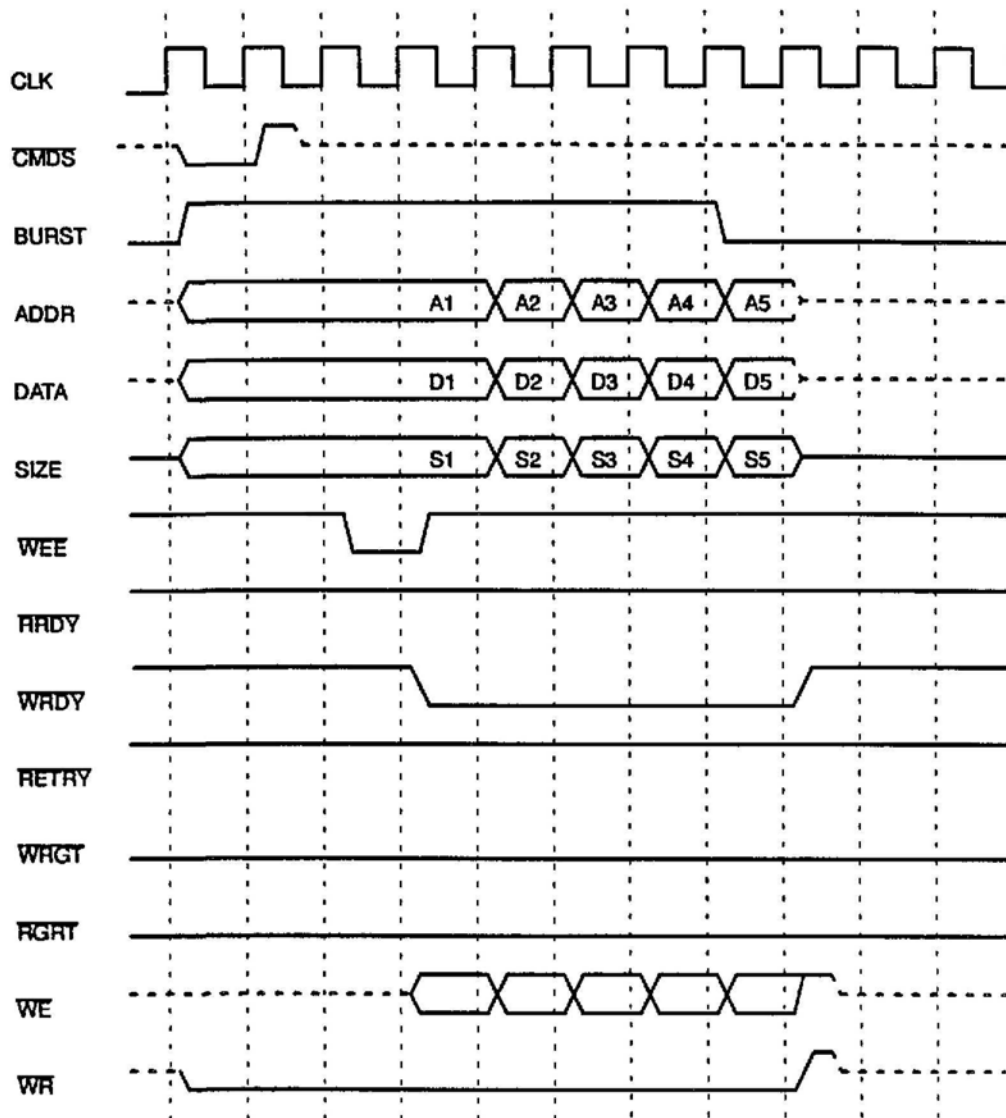
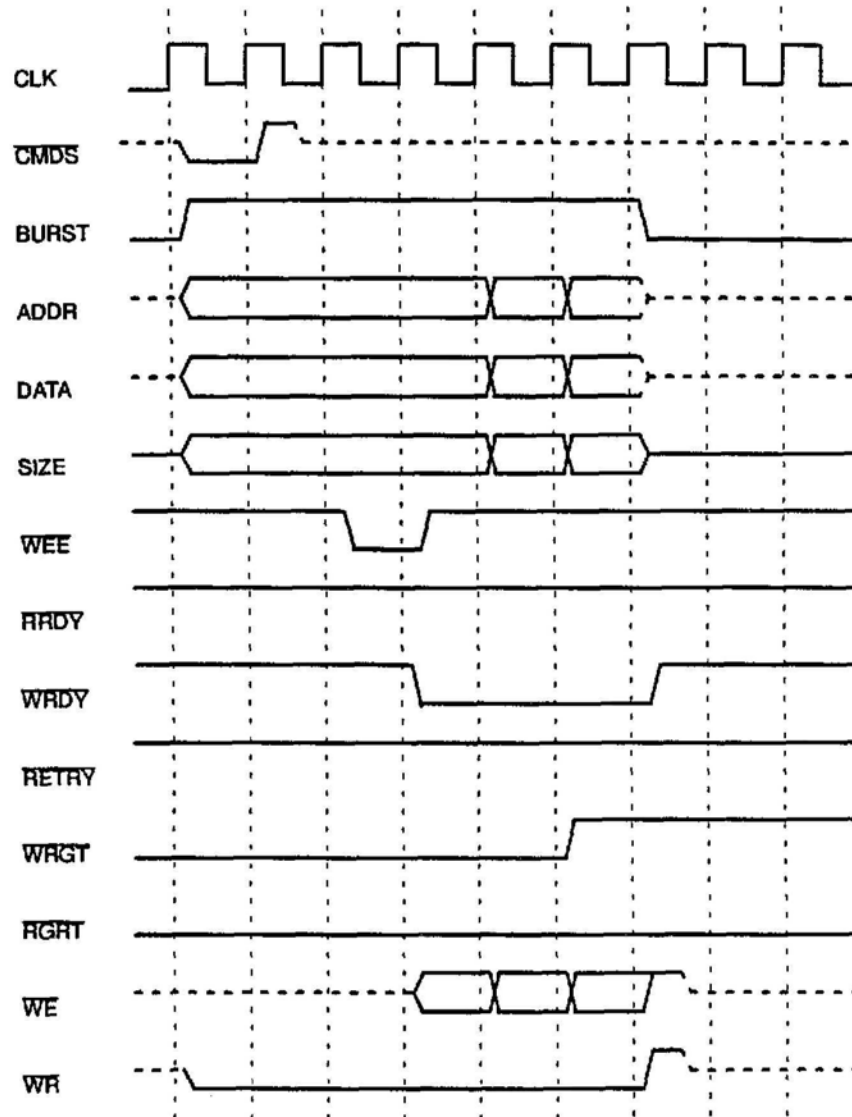


Figure 18–13. VBus-Cacheable Burst Write Hit Abort



### **18.3.8 Shared Write**

Figure 18-14 shows a burst write to shared data. The MXCC does not assert **WEE**, thus preventing the SSP from writing to SRAM. This signal sequence also interrupts the burst cycle. The SSP considers the cycle over; the MXCC assumes actual responsibility to update SRAM (with the data provided by SuperSPARC at **CMDS**) after it has completed a shared write command to the system bus. In MBus configurations, the shared write on the system bus is actually a Coherent Invalidate (CI) that has the effect of making the MXCC cache block exclusive (subsequent cycles to the same block won't be shared). In XBus configurations, a shared write is sent on the system bus, and it is possible that each of the individual addresses in the VBus burst write causes a shared write cycle (i.e., the cache block may not become exclusive).

Figure 18-14. VBus Shared Write

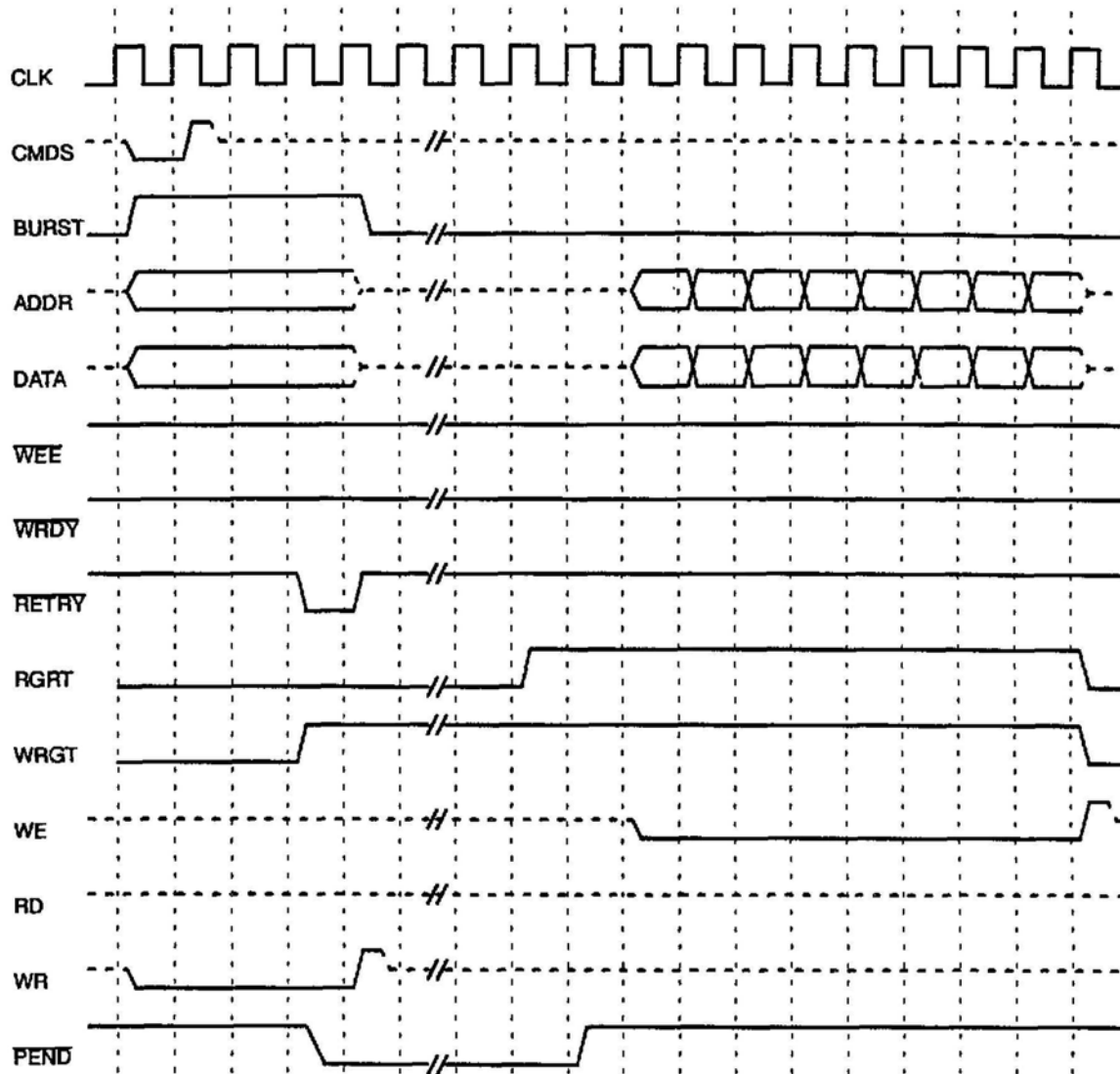
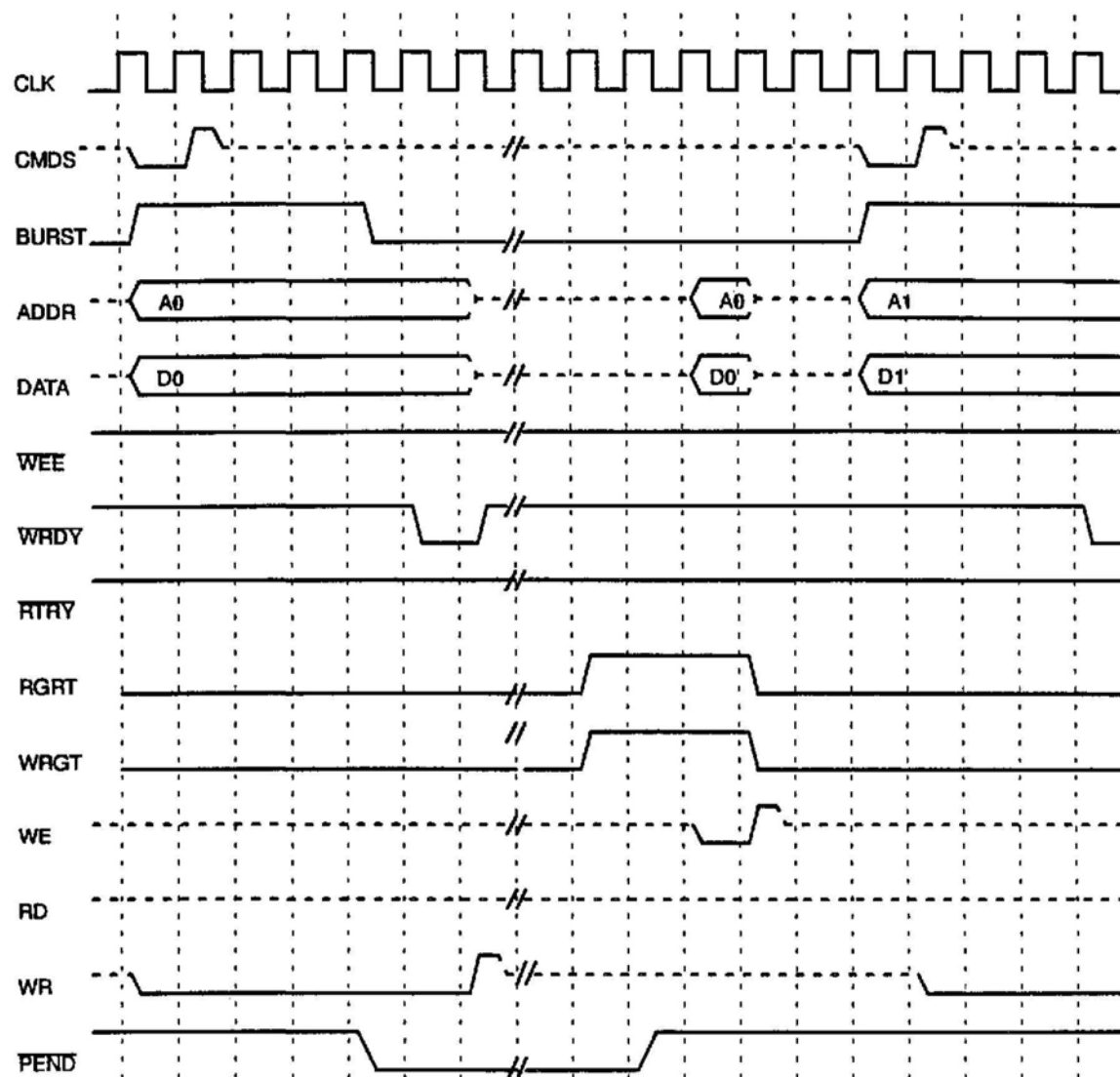




Figure 18–14. VBus Shared Write (Continued)



### 18.3.9 Cache-Disable Write or Non-Cacheable write

Figure 18–15 shows a cache-disable write. The MXCC terminates the VBus cycle by issuing a **WRDY** without asserting **WEE**. A non-cacheable write would be identical.

Figure 18–15. VBus Cache-Disable (or Non-Cacheable) Write

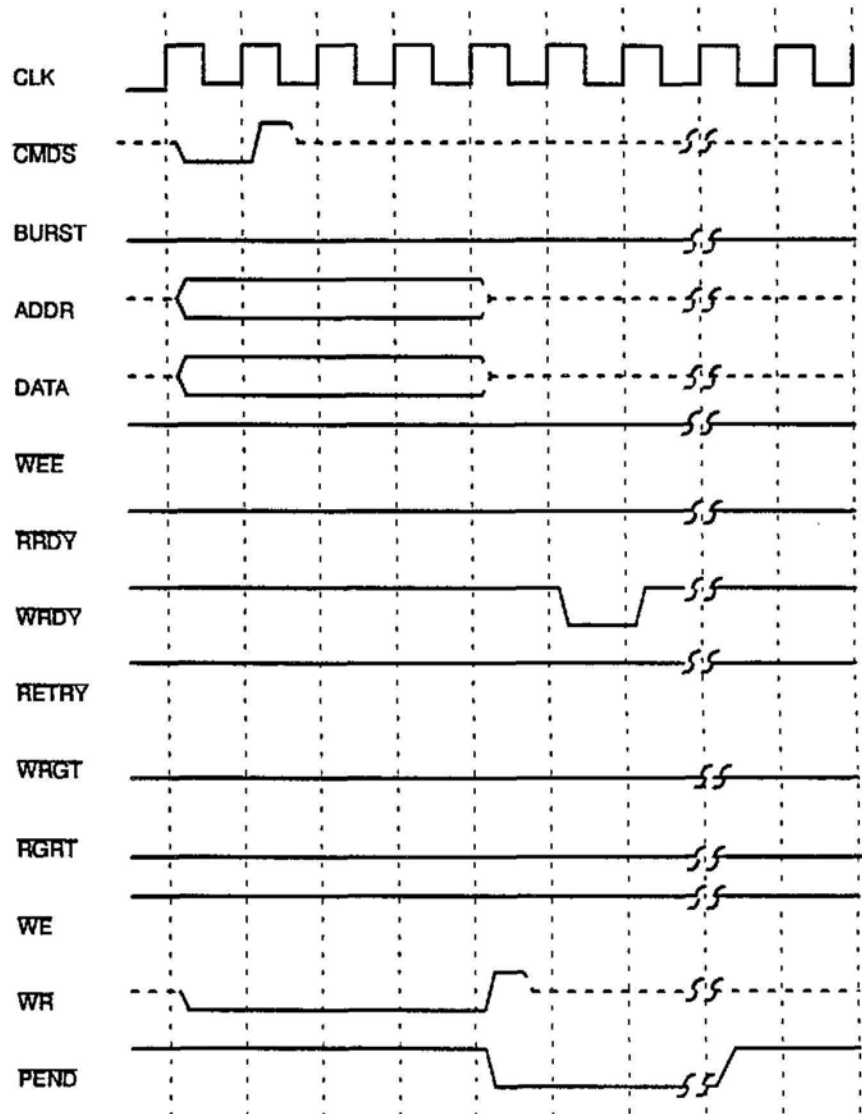
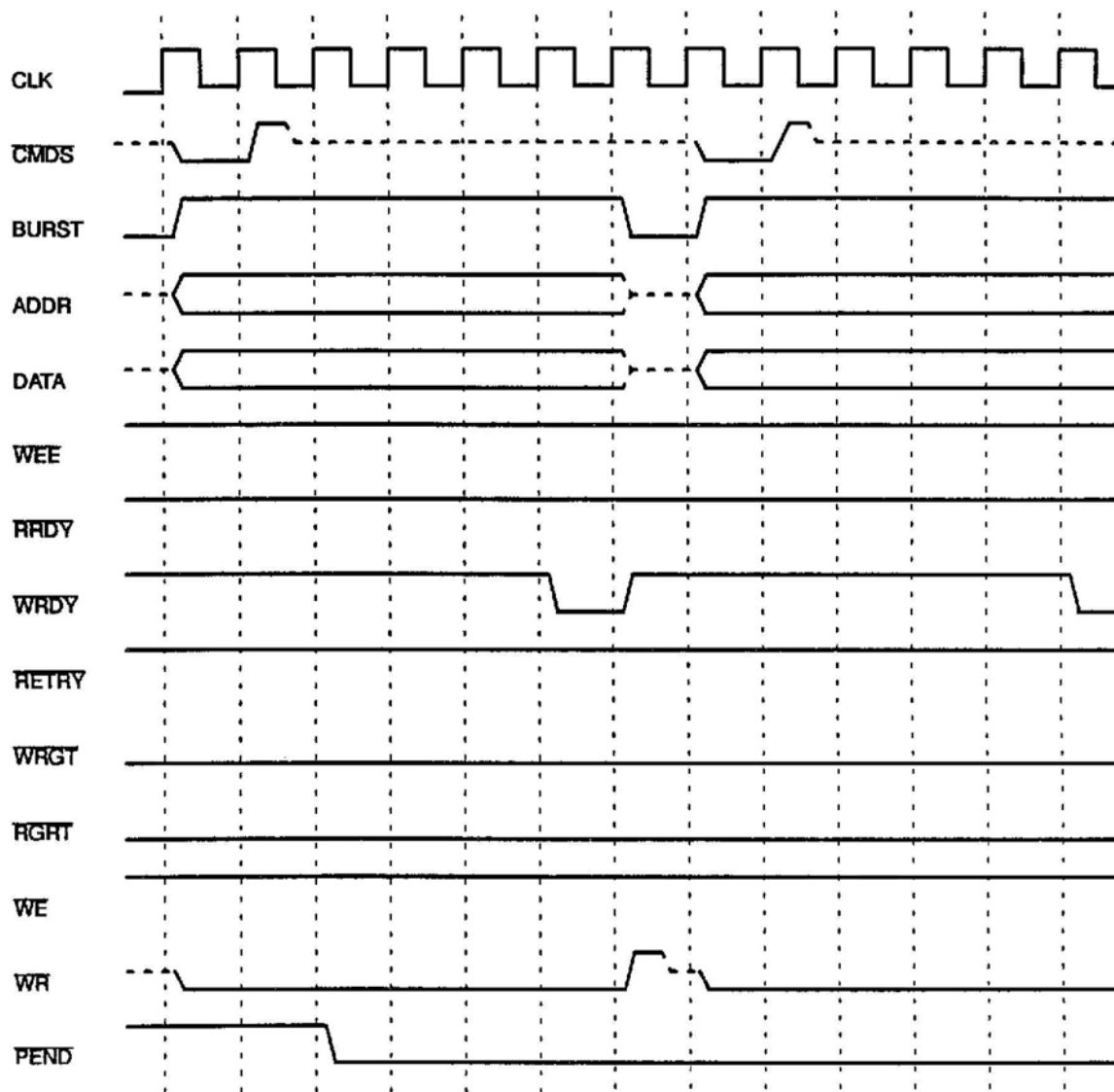


Figure 18-16 shows a burst write cycle when the cache is disabled. The MXCC terminates the cycle and interrupts the burst operation by terminating with a WRDY without asserting WEE.

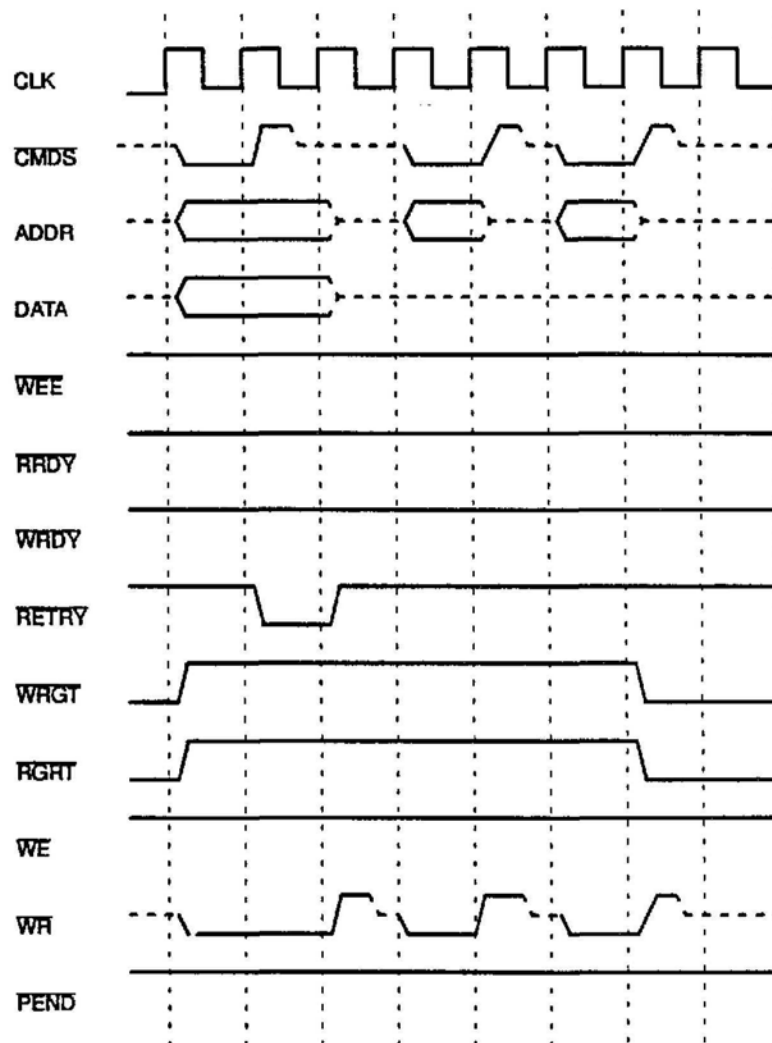
Figure 18-16. VBus Cache-Disable Burst Write



### 18.3.10 Invalidate

Figure 18–17 shows an invalidate. The MXCC first removes the SSP from the VBus by revoking the **RGRT** and **WGRT** bus grants; it then asserts the address, **WR**, and **CMDS**. Multiple Invalidates may occur consecutively. Invalidates may also occur when the MXCC has obtained the VBus for SRAM reads or writes.

Figure 18–17. VBus Invalidation



### 18.3.11 Demap Transactions

Demap transactions are used to remove virtual address translations from the MMUs of processors on VBus and to communicate the removal of translations from the local processor to the rest of the system. MBus does not support demap operations on the system bus—locally initiated demap operations on VBus are ignored by the MXCC and the MXCC never generates demap operations on VBus.

The information transmitted for a demap transaction includes virtual address and type, which the MMU uses as criteria to match pages in the TLB for removal. The context to be used for the demap is broadcast in bits 47 through 32. The lower 32 bits are equivalent to the data format of the demap operation (see Subsection 8.8.2). The exact format of the DATA[63:0] signals is shown in Figure 18–18. Bit 11, bits 0 through 7, and bits 48 through 63 are reserved.

Figure 18–18. VBus Demap Data Format

reserved	Context	VDA	r	Type	reserved1
63	48 47	32 31	12 11	10 8 7	0

<b>reserved</b>	Unused. Sourced by the SSP as zero.
<b>CONTEXT</b>	Context to demap. The context field is compared with the context portion of the TLB entry tag in all participating MMUs. The match is used according to the demap type.
<b>VDA</b>	Virtual demap address. All or part of the VDA field is compared to the virtual address portion of the TLB entry tag in all participating MMUs. The match is used according to the demap type.
<b>TYPE</b>	Demap type. The demap type is encoded the same as MMU demap requests and shown in Table 18–6.
<b>r, reserved1</b>	Unused. Sourced by the SSP as zero.

The TYPE field controls the operation of the demap. Any TLB entries in all participating MMUs that match the hit criteria for the particular demap type should be invalidated. Table 18–6 shows the demap hit criteria for bus demaps. A demap request must meet the access, virtual address, and level criteria of a TLB entry for the type of the demap request for the entry to be a demap hit. Multiple entries in a cache may hit within a single cache for demap types other than page.

Table 18–6. Demap Hit Criteria for Bus Demap Operations

Type	Size	Must meet all three criteria		
		Access	Virtual Address	Level
0	Page – 4K-byte	CONTEXT = context or ACC = 6 or ACC = 7	Tag.Addr = VDA in [31:12]	level = 3
1	Segment – 256K-byte	CONTEXT = context or ACC = 6 or ACC = 7	Tag.Addr = VDA in [31:18]	level = 3 or 2
2	Region – 16M-byte	CONTEXT = context or ACC = 6 or ACC = 7	Tag.Addr = VDA in [31:24]	level = 3 or 2 or 1
3	Context – 4G-byte	CONTEXT = context and ACC < 6	always hit	
4	Entire	always hit		

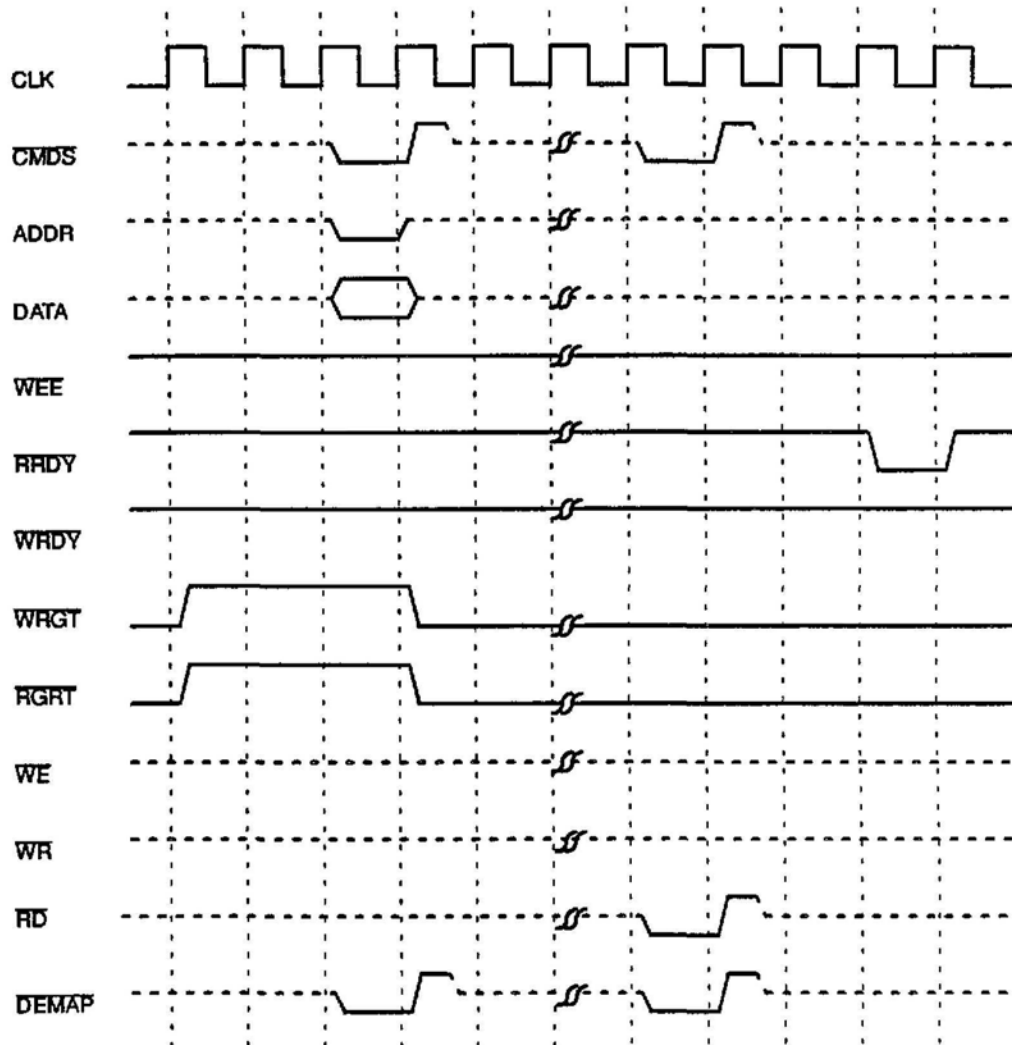
In addition to broadcasting this demap transaction to the rest of the system, the SSP can receive demap transactions originating elsewhere in the system. When a demap is received, the processor executes the demap as if it had been generated internally, but, instead of the current internal context, uses the provided context. The SSP requires a single ready reply for the demap operation. It is the system hardware's responsibility to ensure that the demap is broadcast to and completed by all MMUs in the system. Incoming demaps use a two-phase request/reply protocol.

**Note:**

If broadcast demaps are used, only a single Demap transaction may be pending in the system at any one time. System software is responsible for maintaining this. Indeterminate operation may result if multiple demaps are in progress at the same time by any processor. See Subsection 7.5.3.

Figure 18-19 shows a demap operation on VBus initiated from the MXCC in response to an XBus demap request packet. The MXCC never issues demap operations on VBus if it is in the MBus configuration.

Figure 18-19. VBus MXCC-Initiated Demap

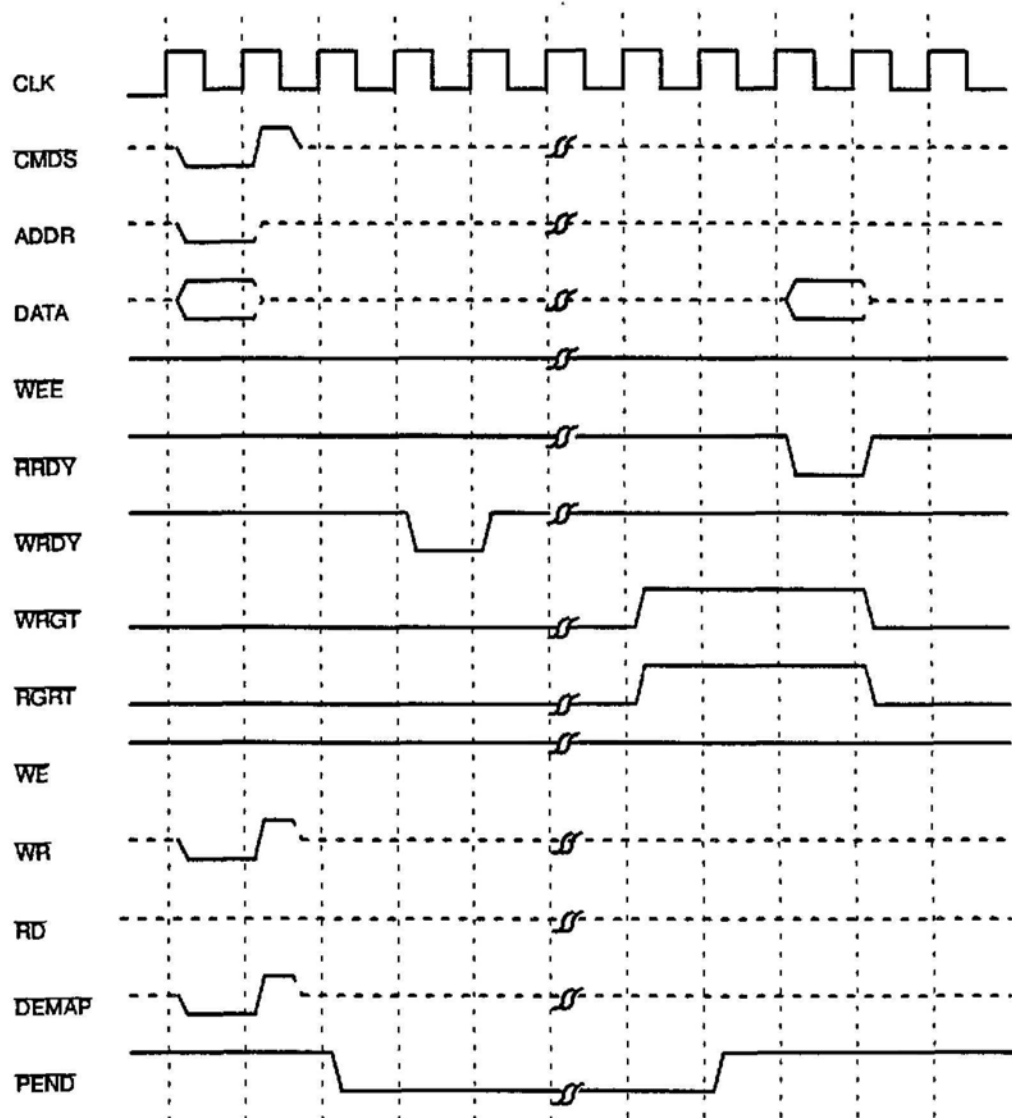


The first phase is the external Demap Request. The MXCC obtains the bus and then asserts **CMDS** and **DEMAP** and drives demap data according to Figure 18-18 onto the data bus. The MXCC supplies an address of zero on the **ADDR** lines.

The second phase is the reply to that request. The SSP asserts **CMDS**, **RD**, and **DEMAP**. The MXCC responds by asserting **RRDY**. There may be other bus activity between the request and reply.

Figure 18-20 shows a demap initiated by the local SSP. To initiate the demap, the SSP asserts **WR** and **DEMAP** with **CMDS** and the demap data in the format shown in Figure 18-18. The **ADDR** signals are all zero during this cycle. There are two replies by the MXCC to this request. The first reply (**WRDY**) acknowledges receipt of the demap request. The second reply informs the processor that the demap has successfully completed across the system and is signalled with the **RRDY** signal.

Figure 18-20. VBus SuperSPARC-Initiated Demap





The MXCC may reply to a demap requests with responses other than WRDY. If the reply is RETRY, the demap operation will be retried on VBus until it succeeds or fails. If the reply is one of the error replies, the error will be reported to the STA instruction that generated the DEMAP just as it would for any other memory access instruction.

### 18.3.12 LDST (Load and Store)

Figure 18-21 shows a load and store instruction to an exclusively owned block. The SSP provides the data to be written beginning at command strobe. The MXCC saves this data, asserts the  $\overline{\text{WRDY}}$  to the SSP, and asserts the  $\overline{\text{OE}}$  to the SRAM to read the current data. Two cycles after the  $\overline{\text{OE}}$ , the MXCC asserts  $\overline{\text{RRDY}}$  as the SRAM provides the current data. The MXCC then completes the write operation to the SRAM.

Figure 18-21. VBus LDST Exclusive Hit

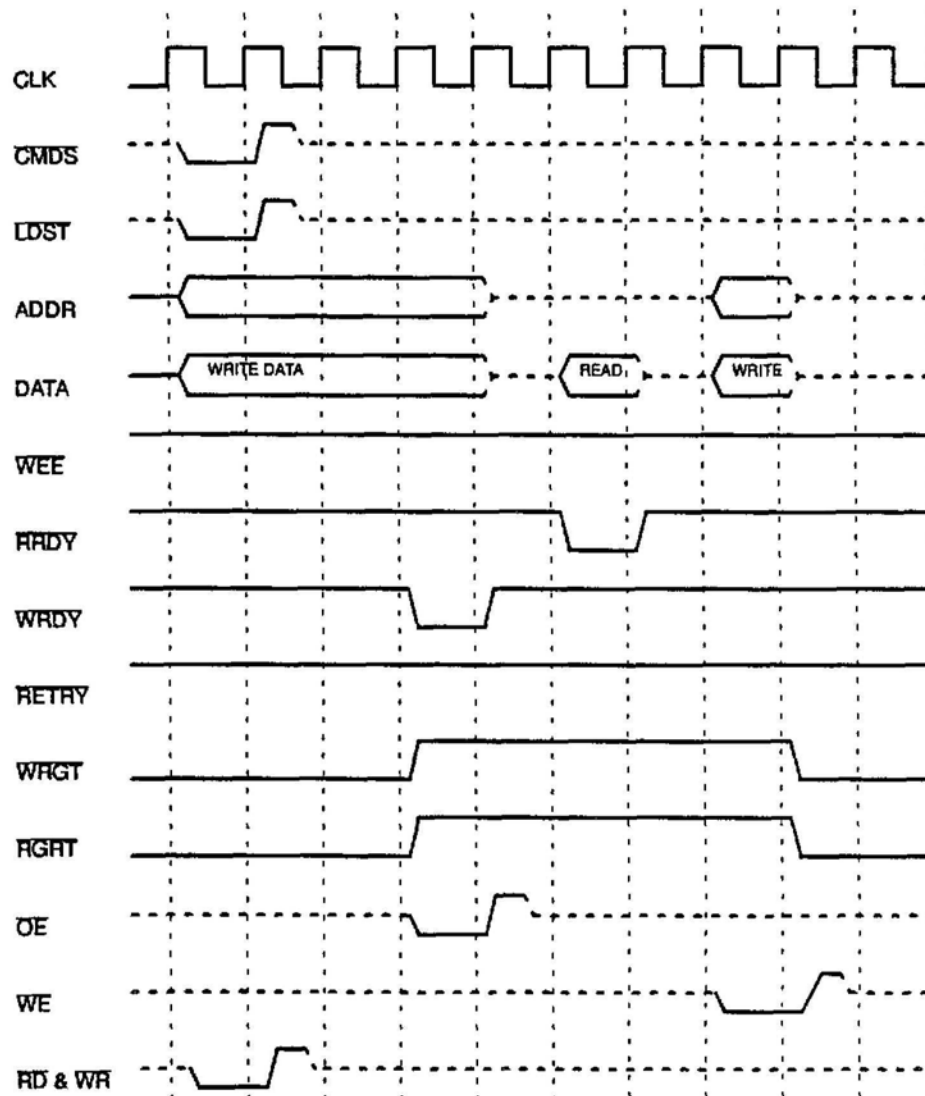


Figure 18-22 shows the same operation when the data is shared. The MXCC asserts **WRDY** to signal that the write data has been saved. The MXCC issues a system cycle (normally **CI** on the MBus and shared write on the XBus). When the system cycle is over, the MXCC asserts **OE**, then asserts the **RRDY** two clocks later as data is provided by SRAMs on the Data bus. The MXCC then writes the write data into the SRAMs.

Figure 18-22. VBus LDST Shared

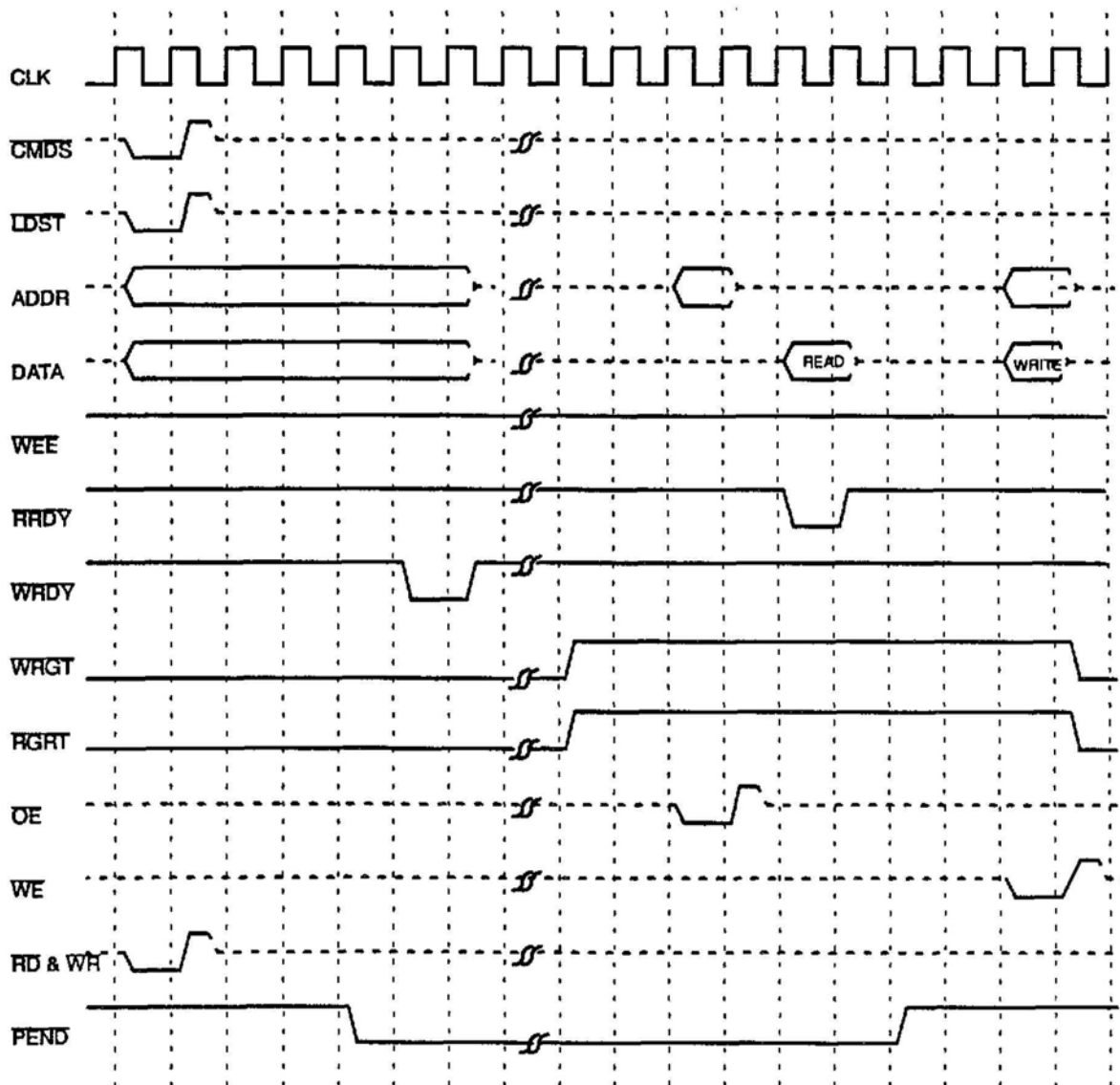
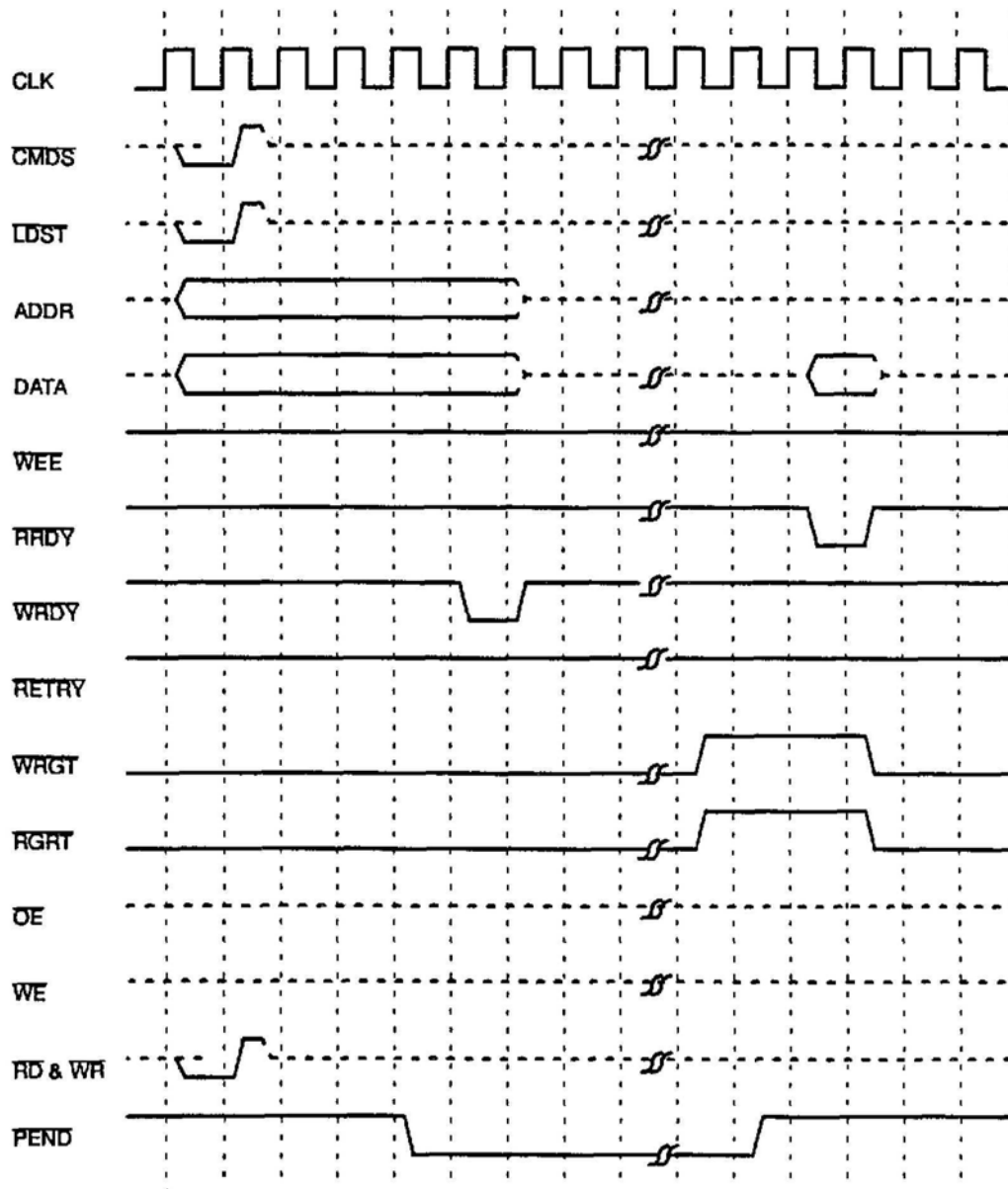


Figure 18-23 shows a non-cacheable load and store operation. In MBus configurations, a locked read/write sequence is executed. In XBus configurations, a swap single request packet is sent on the XBus.

Figure 18-23. VBus Non-Cacheable LDST



### 18.3.13 VBus SRAM and Register Reads

Figure 18-24 shows the SSP reading the SRAM via control space. Figure 18-25 shows the SSP reading the internal the MXCC registers in control space. Figure 18-26 is a write cycle to the MXCC registers.

Figure 18-24. VBus External Cache Read

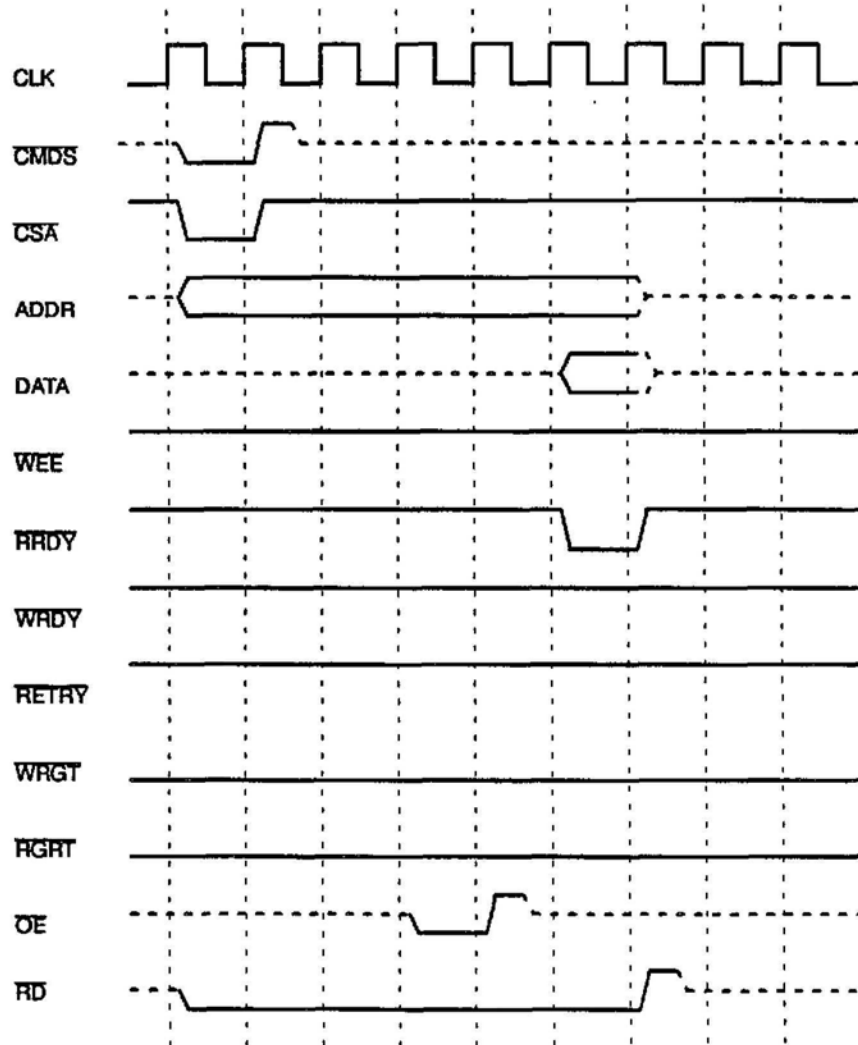


Figure 18–25. VBus Register Read

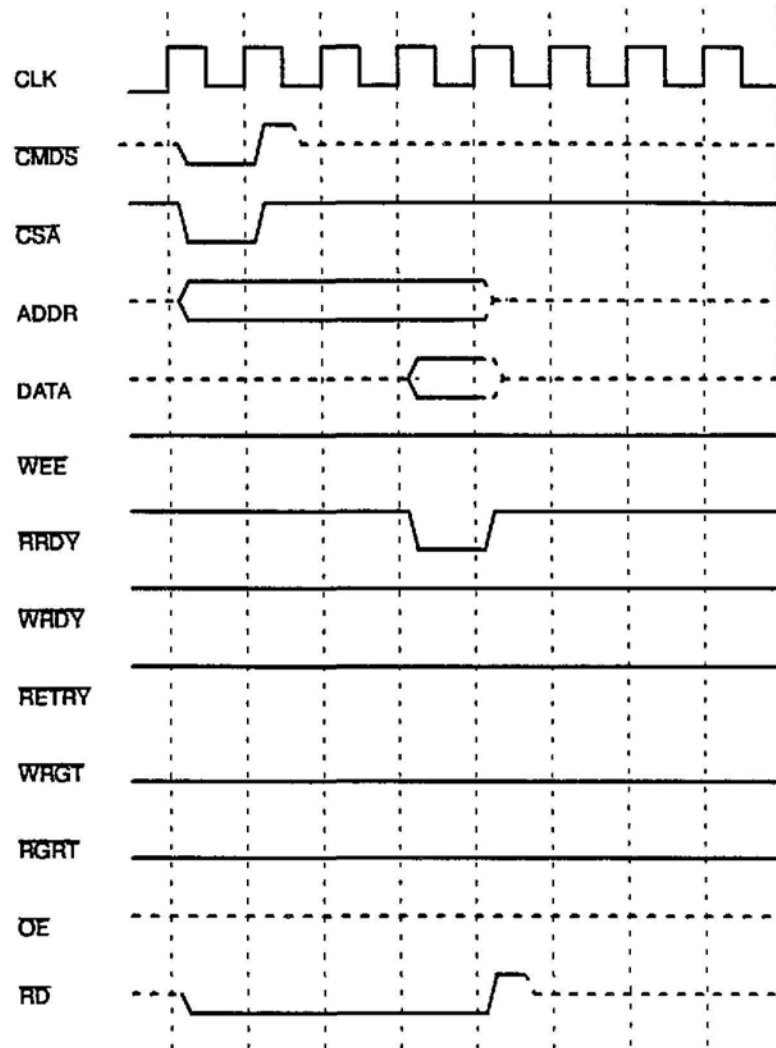
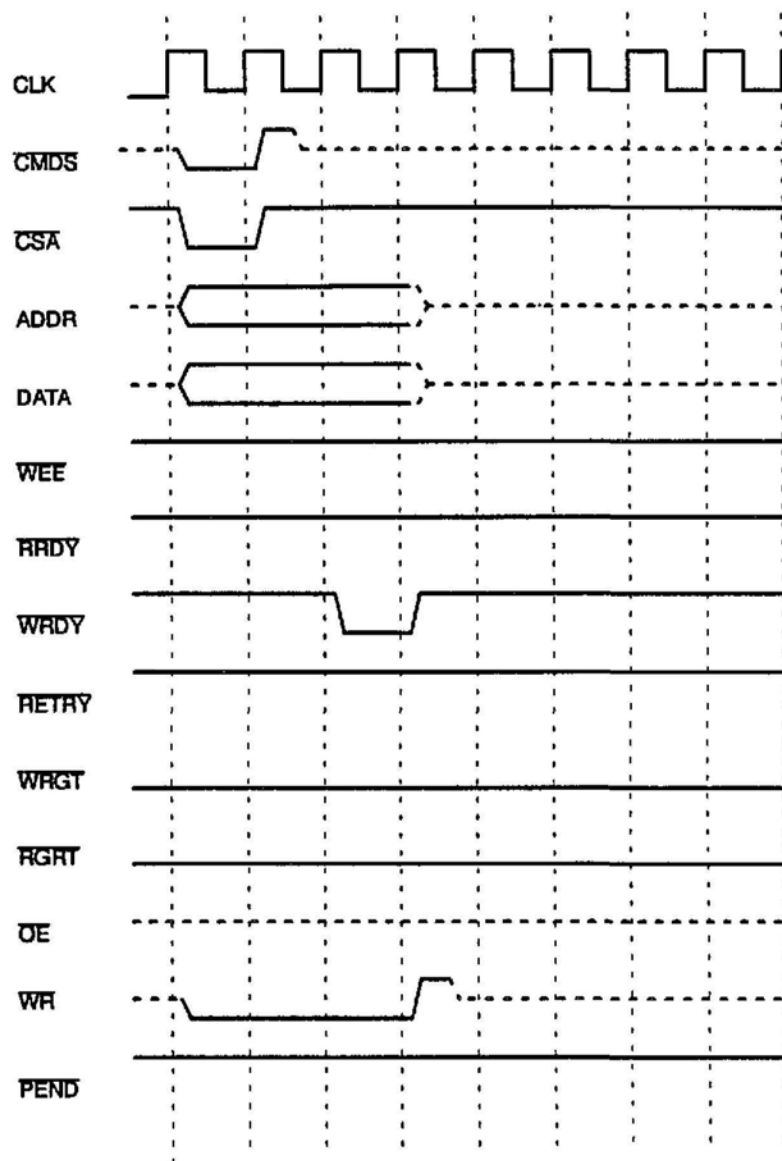


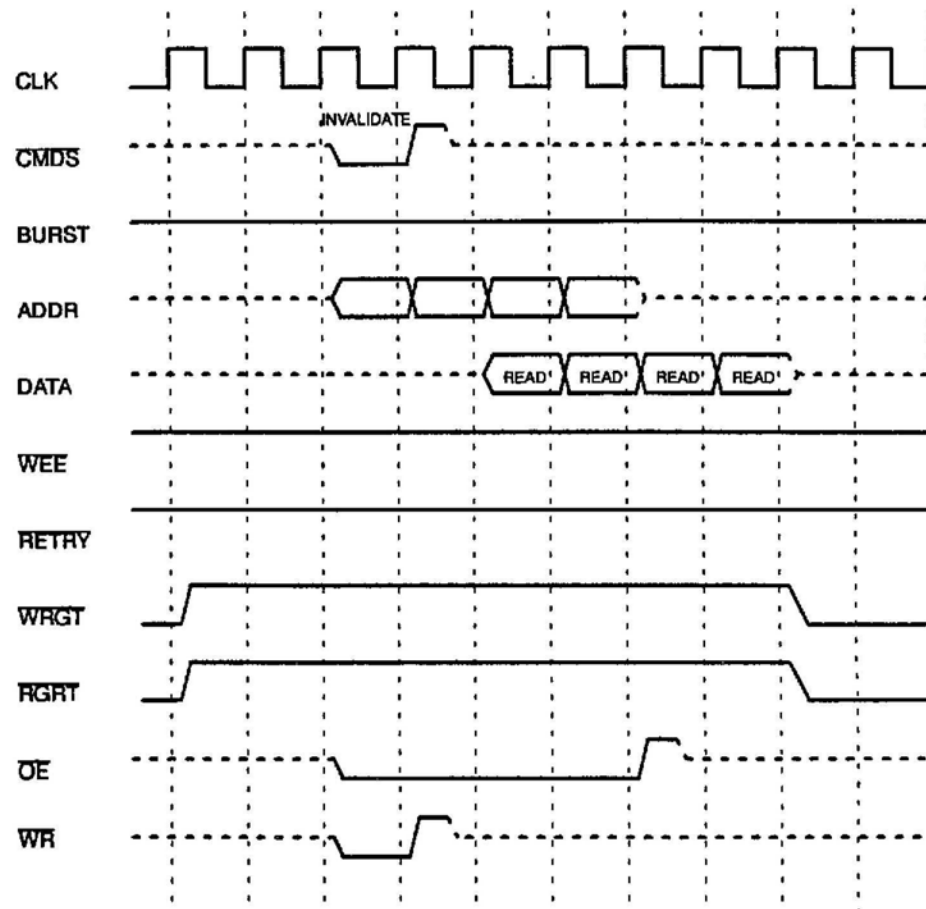
Figure 18–26. VBus Register Write



### 18.3.14 MXCC SRAM Access

Figure 18–27 shows the MXCC removing grants to SuperSPARC and reading the SRAMs. The cycle shown is a Coherent Read and Invalidate (CRI) on the VBus. This CRI causes the MXCC to issue an invalidate to SuperSPARC as the MXCC reads the SRAMs.

Figure 18–27. VBus MXCC SRAM Read



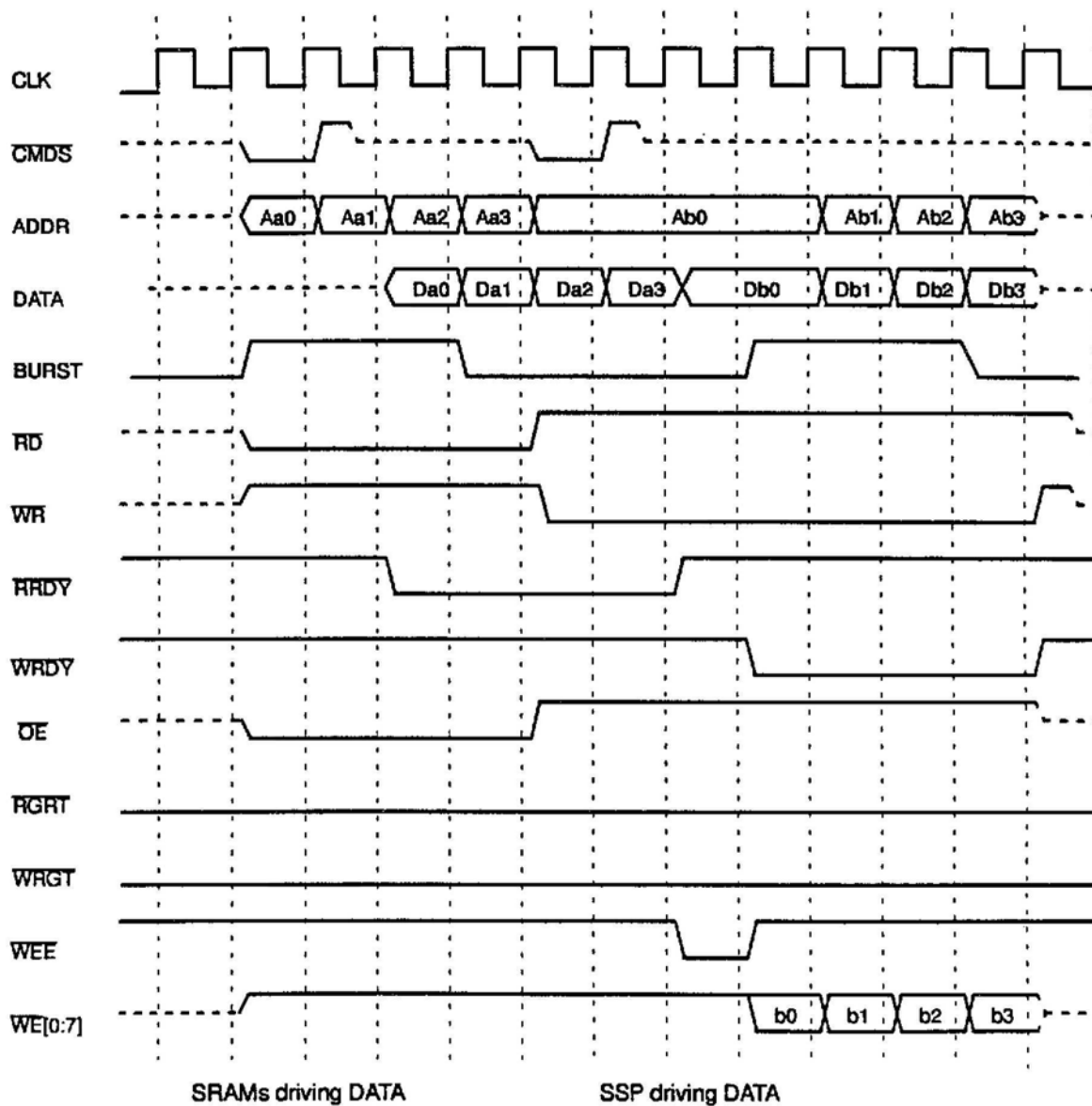
### 18.3.15 Data Bus Contention Avoidance

When there is a read on VBus, followed immediately by a write, the pipelined SRAMs may still be processing the read when the write appears on the address and command lines. If the data lines are also driven for the write in the same cycle, the result will be a driver clash, with drivers in the SRAM trying to drive the SRAM read data onto the VBus at the same time that the SSP is trying to drive the write data onto the VBus.



To prevent this driver clash, the SSP monitors the  $\overline{OE}$  signal and does not drive the VBus DATA[63:0] and DPAR[0:7] signals until at least the third cycle after  $\overline{OE}$  has been sampled as asserted. Figure 18–28 illustrates the action of  $\overline{OE}$ .

Figure 18–28. Action of  $\overline{OE}$  in Preventing Data Bus Driver Clash



# **XBus**

XBus is an extension bus that allows MXCC to be connected to one or more system bus interfaces, called bus watchers. XBus uses an advanced, synchronous, packet-switched protocol to provide low latency and high bandwidth. XBus consists of 82 bussed signals, along with three point-to-point arbitration signals per bus watcher.

<b>Topic</b>	<b>Page</b>
<b>19.1 XBus Overview .....</b>	<b>19-2</b>
<b>19.2 XBus Features .....</b>	<b>19-3</b>
<b>19.3 XBus Signals .....</b>	<b>19-7</b>
<b>19.4 XBus Basic Operation .....</b>	<b>19-9</b>
<b>19.5 XBus Protocol .....</b>	<b>19-15</b>
<b>19.6 Cache Consistency Protocols .....</b>	<b>19-21</b>
<b>19.7 XBus Commands .....</b>	<b>19-23</b>
<b>19.8 MXCC Use of XBus .....</b>	<b>19-35</b>
<b>19.9 Write Hits in Stream Transfers .....</b>	<b>19-40</b>
<b>19.10 Arbitration and Flow Control .....</b>	<b>19-41</b>
<b>19.11 XBus Cycle Waveforms .....</b>	<b>19-49</b>

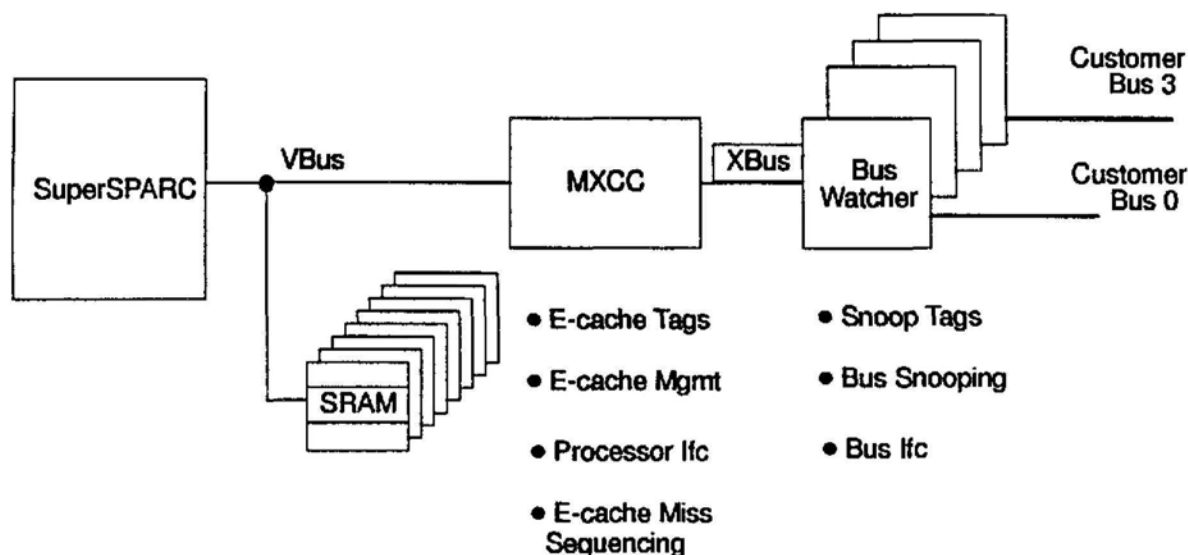
## 19.1 XBus Overview

XBus (Figure 19-1) is designed as a clean dividing line between the functions of the external cache controller that are implemented in the MultiCache Controller (MXCC) and the functions of a multiprocessor bus interface that are implemented in a bus watcher (BW). XBus serves to communicate data requests from the processor and tag state changes from the cache controller to one or more BW's and snoop requests and tag state changes from the BW's to the cache controller.

In this environment, the needs of performance dictate that the cache tags be duplicated so that bus snooping and processor accesses can each be performed with low latency. XBus provides means for keeping duplicate tags in the cache controller and the BW. The BW can initiate any change in cache tag state, allowing proper operation with almost any multiprocessor cache-coherency scheme as dictated by the needs of the backplane bus.

XBus is adaptable to many multiprocessor backplane buses when used with an appropriate BW. The BW is envisioned to be a customer-designed semi-custom device (ASIC) that adapts from the XBus protocol to the particular backplane bus of the customer's system. BWs might adapt XBus to Futurebus+ or other modern high-performance buses.

Figure 19-1. XBus System



## **19.2 XBus Features**

XBus supports high-performance computing through a number of features:

- ☐ Packet-switched bus for high bandwidth and multiple outstanding operations.
- ☐ Division of function between cache controller functions in MXCC and bus interface functions in BWs.
- ☐ Flexible support for cache-coherent multiprocessing.
- ☐ Low-power, high-speed gunning transceiver logic (GTL) electrical interface.
- ☐ Support for up to four BWs per processor.
- ☐ Synchronous or asynchronous operation with processor clock using MXCC's asynchronous interface mode.
- ☐ Separate 36-bit memory address space and 36-bit I/O address space.

### 19.2.1 Packet-Switched Bus

The XBus is a packet-switched (split-cycle) bus. Unlike circuit-switched busses, packet-switched busses allow the same wires to have a higher bandwidth than a similar circuit-switched bus. This is possible because, on a packet-switched bus, the bus wires are free for use between the initial request of an operation and the reply. For a memory read access, this free time on the bus corresponds to the access latency of the memory, a time that can take many cycles on a high-speed bus.

In a circuit-switched bus a bus master (e.g., a processor) that needs to use a slave (such as memory) arbitrates for the bus and obtains ownership. It supplies a slave address and waits for a response. The slave either accepts or supplies data and signals the master when it is done. The master then releases the bus.

In a packet-switched bus a processor that needs to use memory also acquires the bus. But on a packet-switched bus, the processor keeps the bus just long enough to send a request message consisting of a target I.D., a command, its own return I.D., and data (if appropriate). After sending the message, the processor releases the bus. Each resource on the bus (such as memory subsystem) monitors messages looking for messages targeted to it. When a bus resource detects such a message, it attempts to obey the command. When the resource completes the requested operation, it signals completion by acquiring the bus with the same arbitration mechanism and then sending a completion message back to the originator of the command. The completion message contains the ID of the slave, the requestor's ID, status information, and data (as appropriate).

Broadcast messages (intended for multiple destinations) are also possible with packet-switched busses. Each addressed resource can return a completion message in turn by arbitrating for the bus.

### 19.2.2 Division of Functions

XBus serves to separate cache functions as implemented in the MXCC from bus interface functions as provided by the BW. By dividing these functions, the MXCC can be used with a wide variety of system interconnects, both bussed and non-bussed.

#### ☐ MXCC

The MXCC is responsible for the following functions:

- E-cache tags and control,
- E-cache miss handling,
- Prefetch handling,
- XBus arbitration,

- Block memory copy/zero,
  - Interrupt handling,
  - Synchronization for operating the bus at a clock rate different from the processor's, and
  - BootBus.
- ☐ BW
- A BW is responsible for the following functions:
- Interfacing to the system interconnect (includes protocol, data formatting, access control, electrical protocol, etc.),
  - Cache coherence protocol (includes snooping of the system bus or participation in directory-based protocols),
  - Coordinating system-wide completion of demap requests,
  - Containing system resources that are duplicated per processor, and
  - Snooping its own operations.

### 19.2.3 System Configuration

XBus offers flexible cache-coherent multiprocessing. The division of functions allows flexibility in configuring systems. Different BW designs may be devised to match a wide variety of system organizations and a wide variety of system interconnects. The cache-coherence protocol is controlled by the BW and XBus. The MXCC can be used with any protocol that can be mapped to the cache block states of the E-cache tags in the MXCC.

The duplicate sets of cache tags in the MXCC and in the BWs must be kept consistent. In order to prevent momentary inconsistencies from resulting in a loss of system consistency, the BW tags are always updated first. The BW updates its snoop tags for each operation it receives from the MXCC or from the bus. When the snoop tags change, the E-cache tags are updated using the TCmd field of a request or reply packet to the MXCC.

### 19.2.4 Electrical Interface

XBus operates with GTL electrical interface. GTL is a low-voltage swing signaling scheme for high frequencies, low power, and low cost. When using XBus and GTL levels, the GTL reference voltage (GTL-REF) must be supplied to the MXCC and the BWs.

### 19.2.5 Bus Interfaces

MXCC has direct support for one, two, or four BWs. This allows several system buses to be used to provide more bandwidth for system interconnect.

The built-in XBus arbiter in the MXCC provides arbitration for one, two, or four BWs.

### **19.2.6 Dual Clocks**

The MXCC's support for synchronous or asynchronous operation of the processor and the bus may be utilized in XBus systems. When asynchronous operation is selected, the processor clock may be faster than the XBus clock.

### **19.2.7 Address Spaces**

XBus has two sets of commands that access two separate address spaces:

☐ **Memory Address Space**

This is a byte-addressed 36-bit, cache-coherent address space.

☐ **I/O Address Space**

This address space is also 36-bits. It is not cached by the MXCC.

### **19.2.8 Bus Watchers**

BWs interface XBus to application-specific system buses or devices. Their function is to translate:

☐ XBus transactions into system bus or device operations, or

☐ System bus or device operations into XBus transactions.

At the lowest operational level, BWs:

☐ Receive XBus packets that request system bus resources or system bus actions and map them to the appropriate system bus commands.

☐ Receive system bus responses or replies to these requests and map them to XBus reply messages.

☐ Receive system bus commands directed to the XBus and convert them to appropriate XBus command packets.

☐ Receive XBus replies and map them to corresponding system bus responses.

☐ Snoop system bus operations for references to locally cached data and send messages to the MXCC to perform coherency operations.

## 19.3 XBus Signals

XBus signals are divided into the following groups:

- ☐ Control,
- ☐ Arbitration, and
- ☐ Data

Except for the clock and data signals, all signals are encoded low true and are written with an overbar to indicate negated logical values.

### 19.3.1 Control Signals

The control group contains the bus clock and an error signal.

<b>XCLK</b>	Provides all timing for XBus signals. XCLK is an input to all devices on an XBus.
<b><math>\overline{\text{CCErr}}</math></b>	Used by MXCC to indicate an unrecoverable error in MXCC or the SuperSPARC processor (SSP) to the BWs. Always driven by MXCC. Bussed to all BWs.

### 19.3.2 Arbitration Signals

The arbitration group contains two request signals and a grant signal per device.

<b>XREQ0[1:0]</b>	Bus request and control flow from BW0 to MXCC.
<b>XREQ1[1:0]</b>	Bus request and control flow from BW1 to MXCC.
<b>XREQ2[1:0]</b>	Bus request and control flow from BW2 to MXCC.
<b>XREQ3[1:0]</b>	Bus request and control flow from BW3 to MXCC.
<b>XGNT0</b>	Bus grant from MXCC to BW0.
<b>XGNT1</b>	Bus grant from MXCC to BW1.
<b>XGNT2</b>	Bus grant from MXCC to BW2.
<b>XGNT3</b>	Bus grant from MXCC to BW3.



**Notes:**

An  $\overline{XGNTn}$  signal is asserted continuously for the number of cycles granted. Its duration is two cycles or nine cycles, depending on the length requested (see Section 19.4).

All signals in the arbitration group are point-to-point and are always driven.

### 19.3.3 Data Signals

The data group contains the 64 data signals and four parity signals.

**XData[63:0]** These bussed bidirectional signals carry the information being transported on XBus as well as the headers that contain address and control information. A device drives XData only after receiving  $\overline{XGNT}$  from the arbiter.

**XPar[3:0]** These bussed bidirectional signals carry parity computed over for XData. The parity for a given value of XData appears in the same cycle as the value. The XData signals checked by each of the XPar signals are:

- XPar[3] checks XData[63:48]
- XPar[2] checks XData[47:32]
- XPar[1] checks XData[31:16]
- XPar[0] checks XData[15:0]

For each value of XData there are two correct encodings of XPar[3:0]: all even and all odd. In all-even encoding, there are an even number of logic ones in each of the four sets of 17 bits, comprising a 16-bit portion of XData and the XPar bit that checks it. All odd encoding has an odd number of ones in each of the same four sets of 17 bits.

The all-even encoding is used for header cycles, while the all-odd encoding is used for data cycles. Any combination of XData and XPar that is neither all even nor all odd indicates a parity error.

All-even parity during a data cycle is a special indication that the cycle is a memory fault cycle. See Memory Fault Cycle, page 19-20.

## 19.4 XBus Basic Operation

The XBus operation has three levels:

- ☐ Cycles,
- ☐ Packets, and
- ☐ Transactions.

### Cycles

A bus cycle is one period of the clock; it forms the unit of time and one-way information transfer.

### Packets

A packet is a contiguous sequence of cycles that constitutes a unidirectional command and information transfer. XBus packets are either two or nine cycles (one cycle with command and status information and either one or eight cycles of data).

### Transactions

A transaction generally consists of a request and reply pair of packets, but a few transaction types have only a request packet.

Each chip on the XBus can have several XBus sources and targets, each identified by a unique XBus ID. An arbiter permits the bus to be multiplexed among the various chips. Before a chip can send a packet, it must be granted mastership by the arbiter. Once it has control of the bus, it puts the packet on the bus one cycle at a time, without interruption.

A reply packet must be sent to the target ID that was the source ID of the request packet that it answered.

The basic layout of a two-cycle packet is shown in Figure 19-2; the basic layout of a nine-cycle packet is shown in Figure 19-3. Each begins with a header cycle. The packets complete with one or eight data cycles. The interpretation of the data cycles depends on the command in the header cycle.

Figure 19-2. Basic Format of a Two-Cycle Packet

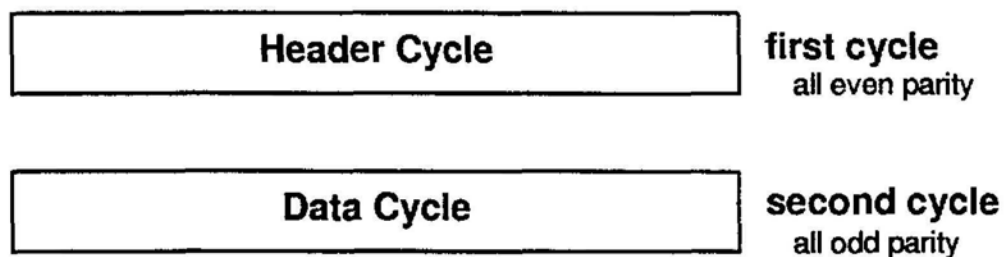
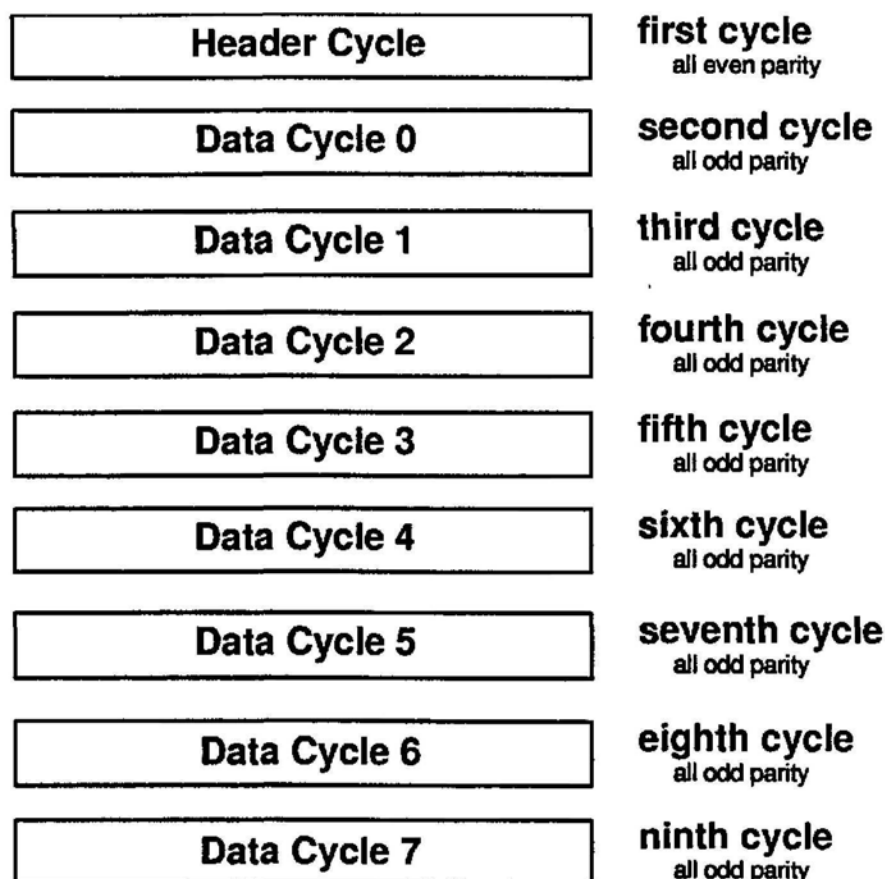


Figure 19–3. Basic Format of a Nine-Cycle Packet



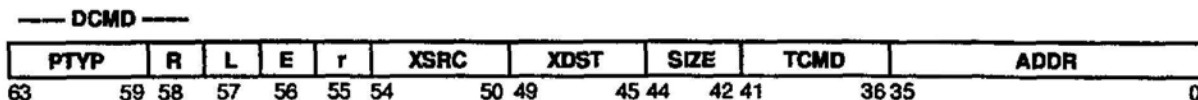
### 19.4.1 Packet Headers

A header cycle is the first cycle of every XBus packet. The header contains data and tag commands, the memory or I/O address involved, and the XBus IDs of the source and destination of the packet.

Header cycles are marked with all even parity on the parity lines ( $XPAR[3:0]$ ). Data cycles have odd parity computed on each of four groups of 16 bits ([63:48], [47:32], [31:16], [15:0]). Devices must use parity checking to help locate packet headers in the stream of cycles on the bus.

The basic format of the header cycle is shown in Figure 19–4.

Figure 19-4. Basic Format of the Header Cycle



The fields of the header cycle are introduced here and will be described in detail later in this chapter.

PTYP	Packet Type. Names the data command to transfer data between BWs and the MXCC.
R	Reply packet. This bit is set on reply packets and clear on request packets.
L	Packet Length. This one-bit field indicates the packet length. (0 = two-cycle; 1 = nine-cycle)
E	Error. In a reply packet, indicates that the corresponding request packet encountered an error.
r	reserved.
XSrc	XBus Source ID.
XDst	XBus Destination ID. Specifies the intended recipient of this packet.
Size	Number of bytes of data that will be transported by this transaction.
TCmd	Tag Command. Specify changes in cache or snoop tags for the specified address.
Address	Physical byte address as needed by the data command (DCmd) and the tag command (TCmd).

The TCmd is used to keep the bus and processor side copies of cache tags consistent with one another. The TCmd and DCmd commands, along with the various control bits, provide sufficient flexibility to accommodate a variety of system busses.

All devices are responsible for monitoring the bus for packets addressed to them. If they see a packet addressed to them, they perform the command contained in the header cycle.

### 19.4.2 Bus Watcher ID Decoding and Addressing

The MXCC sends request packets destined to any BW to a single XDest of 0x10. BWs can recognize packets by the XDest field in the header. BWs use Address[9:8] to partition requests (if present) among the several BWs. Table 19-1 shows how Address bits participate in decoding the BW that is addressed. Only one BW will reply to any request from the MXCC, although, in some systems, all may need to act upon the tag command. All need to act upon broadcast messages.

Table 19-1. BW Addressing

Number of BWs	Address Decode
1	always
2	Address[8] = BW#
4	Address[9:8] = BW#

In addition to sending packets to the MXCC, a BW can send packets onto XBus addressed to itself or to other BWs.

### 19.4.3 XBus Arbitration

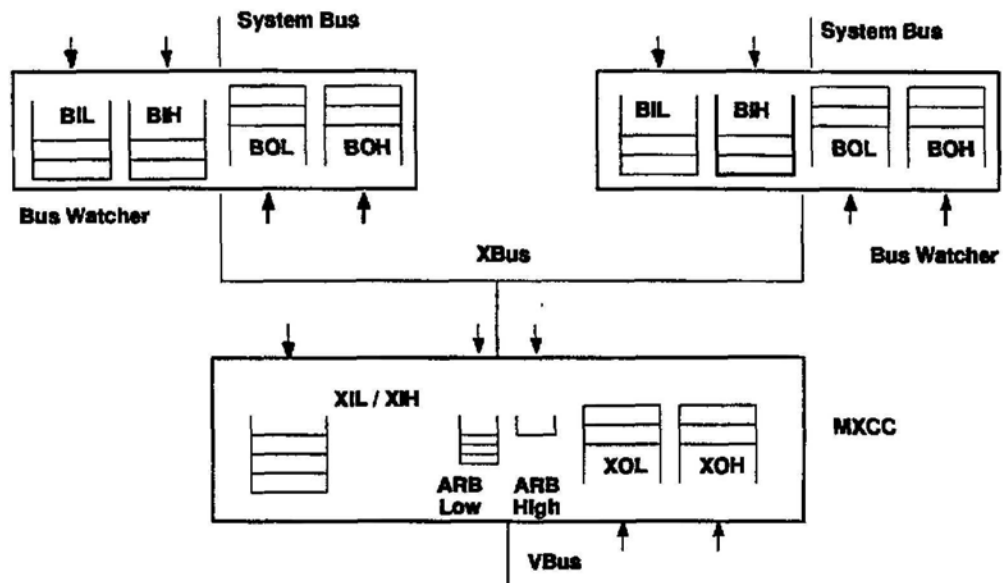
The MXCC contains a pipelined arbiter that controls access to XBus. To use the bus, BW number  $n$  uses the point-to-point  $XREQ_n[1:0]$  lines to request it. The bus request is always for a specific number of bus cycles (either two or nine cycles). The arbiter inside MXCC grants ownership with a point-to-point grant line,  $XGNT_n$ . The grant line is generally asserted for the number of cycles allocated to the requesting device (either two or nine cycles). The device that receives the grant drives the bus for exactly two or nine cycles. See Section 19.10 for details.

### 19.4.4 Packet Priority

Packets from the BWs are categorized as either high-priority or low-priority. Packets given high priority are limited to system bus reply packets resulting from cache read misses on the VBus. All other reply packets and all request packets are given low priority. A BW that has bus grant must send all high-priority packets in its queues before sending any low-priority packets. This priority mechanism allows packets containing data that the SSP is waiting for (such as data to complete a read miss) to bypass packets that are not as critical to the processor (such as prefetch data or requests from the system bus).

To accomplish this priority scheme, the MXCC employs two first-in first-outs (FIFOs) each for receive and transmit packets (see Figure 19-5). Most BW designs would also use dual queues.

Figure 19-5. Queues in Bus Watchers and MXCC



Following are the queues and their explanations for Figure 19-5.

BOL	BW low-priority outgoing queue that contains requests for system bus resources.
BOH	BW high-priority outgoing queue that contains replies to system requests.
BIL	BW low-priority input queue that contains system requests and most system replies.
BIH	BW high-priority input queue that contains read miss replies.
XOL	Low-priority MXCC output queue (requests).
XOH	High-priority MXCC output queue (replies).
XIL/XIH	Low- and high-priority MXCC input queues (actually combined into one queue).
Arb L	Holds low-priority XBus arbitration requests.
Arb H	Holds high-priority XBus requests

#### **19.4.5 XBus Flow Control**

Since the operation of both the BWs and the MXCC depends on queues, it is important that the queues never overflow. The MXCC protects its input queue (XIH/XIL) from overflowing with the XGNT<sub>n</sub> lines. The MXCC does not issue XGNT to any BW when its input queue is too full.

In order to protect its queues (BOH/BOL) from overflowing, a bus watcher uses the XREQ<sub>n</sub> lines to inhibit packets on the XBus from the MXCC (either requests or replies). A BW may also request the XBus and simultaneously inhibit packets coming to it from the MXCC. The following is a list of the commands that can be encoded in the XREQ<sub>n</sub> lines:

- ☐ Inhibit the request packets from the MXCC to the BWs and system bus.
- ☐ Inhibit request and reply packets from the MXCC to the BWs and system bus.
- ☐ Request the XBus to send a low-priority packet from the BW to the MXCC.
- ☐ Request the XBus to send a high-priority packet from the BW to the MXCC.
- ☐ Request from the XBus either a low- or high-priority packet from the BW and also inhibit requests and replies from the MXCC to the BWs.

## 19.5 XBus Protocol

XBus follows a protocol that is understood by all devices. The XBus protocol is defined in terms of cycles, packets, and transactions.

### 19.5.1 Cycles

A bus cycle is one period of the bus clock; it forms the unit of time and one-way information transfer.

All cycles on the XBus fall into one of four categories:

☐ Header

A header cycle is always the first cycle of a packet. The header defines the packet size and conveys the data command, tag command, and address for the packet. Header cycles have all-even parity.

☐ Data

Data cycles normally constitute the remaining cycles of a packet and convey the data that accompanies the command. Data cycles have all-odd parity.

☐ Memfault

MEMFAULT cycles are used to indicate an error in one of the data cycles of a packet. Memfault cycles have all-even parity.

☐ Idle

Idle cycles are those during which no packet is being transmitted on the bus. Idle cycles have all-odd parity.

A given cycle with all even encoding is a HEADER cycle if it is the first cycle of a packet; otherwise, it is a MEMFAULT cycle. A given cycle with all odd encoding is a DATA cycle if it is known to lie inside some packet; otherwise it is an IDLE cycle.

When the parity encoding is neither all-even nor all-odd, the cycle has a transmission parity error.

### 19.5.2 Packets

A packet is a contiguous sequence of cycles. The first cycle (header) of a packet carries address and control information, while subsequent cycles carry data. Packets come in two sizes: two cycles and nine cycles. The size is indicated by the L bit in the header cycle.

An XBus device sends a packet after arbitrating for the XBus and getting grant. Packet transmission by a device is uninterruptable once the header cycle has been sent.



A six-bit DCmd field bit in each packet encodes the packet type. One of these bits (the R bit) encodes whether the packet is a request or a reply; the other five and packet types (PTYPs) and encode the transmission type. Detailed information about various DCmds is provided in Subsection 19.5.5.

### 19.5.3 Transactions

A transaction consists of a pair of packets (a request packet and a reply packet) that together perform a logical function. The PTYP for a reply is the same as for the request to which it is responding. Replies are sent to the source of the request, so the XDST field of a reply is the same as the XSRC from the request.

Packets usually come in pairs, but there are two exceptions to this. For the Flush Block transaction, several reply packets may be generated for one request. Since Flush Block Requests are unilaterally generated within the MXCC, several Flush Block Reply packets may be sent without any request on XBus. For a transaction that times out, no reply packet may be generated.

### 19.5.4 Packet Detection

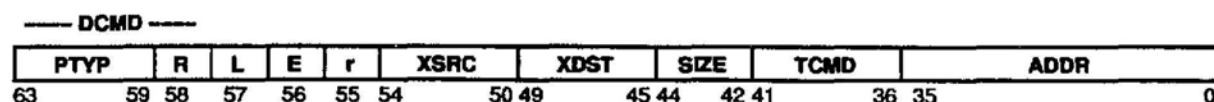
Header cycles are indicated by even parity encoding on each of the four parity bits XPAR[3:0]. The XBus device uses this information as well as its current XBus state information to recognize a header cycle. Each XBus device is expected to track activity on the bus and know whether the current cycle is idle, a header cycle or a data cycle. The purpose of the four parity bits is to confirm the expected bus state rather than to detect it.

Once the header cycle has been recognized, the XBus device expects data. The number of data cycles is determined by the length bit in the message header. Data and idle cycles have the same parity encodings, thereby preventing the parity from distinguishing between them.

### 19.5.5 Header Cycle Format

Header cycles are indicated by the all-even encoding on the parity wires. The format of the header bits is shown in Figure 19-6.

Figure 19-6. XBus Header Cycle Encoding



Bit field explanations:

DCMD

Data Command. This field contains the six-bit data command composed of the five-bit PTYP and 1-bit R fields. Refer to Subsection 19.7.1.

PTYP	Packet Type, a subfield of DCmd. Indicates the type of transaction to which this packet belongs.
R	<p>Reply, a subfield of DCmd. This bit indicates that the packet is a reply rather than a request. The reply to a request has the same DCMD with R = 1.</p> <ul style="list-style-type: none"><li>■ 0: Request packet.</li><li>■ 1: Reply packet.</li></ul>
L	<p>Packet Length. This one-bit field indicates the packet length.</p> <ul style="list-style-type: none"><li>■ 0: Two-cycle packet</li><li>■ 1: Nine-cycle packet</li></ul>
E	<p>Error. The error bit is defined only for reply packets. It indicates that the corresponding request packet encountered an error. The second cycle of an error reply provides additional error information, and the remaining cycles are ignored. Note that the length of an error reply packet is the same as the length of the corresponding normal reply packet. See Subsection 19.5.6 for more information on the ERR field.</p>
r	One-bit reserved field. Value must be zero.
XSRC	XBus Source ID. Used by the MXCC to more fully qualify the DCmd fields of packets it sends to BWs. The MXCC reflects the XSRC field of packets from BWs into the XDST field of replies.
XDST	<p>XBus Destination ID. This field specifies the intended recipient of this packet. All packets sent by the MXCC to BWs have XDST = 0x10. Packets from BWs to the MXCC must have XDST as shown in Table 19-9 and Table 19-10.</p> <p>When the XDST specifies a BW (0x10), additional information such as address bits may be needed to select between the several BWs on this XBus.</p>

**SIZE**

The size field specifies the number of bytes of data that will be transported by the current transaction. See Table 19-2.

*Table 19-2. Size Field Specifications*

SIZE CODE	SIZE
000	1 byte
001	2 bytes
010	4 bytes
011	8 bytes
100	– reserved
101	64 bytes
110	– reserved
111	– reserved

The MXCC never generates packets with reserved size codes in Table 19-2. The MXCC has undefined behavior if it receives a packet with a size code shown as reserved.

**TCMD**

Tag Command. The tag command specifies how the tag entry for the specified address should be manipulated. Refer to 19.7.2.

**ADDR**

Address. The 36-bit address field specifies the byte address. The ordering of bytes is big-endian. References are aligned on the same boundaries as the size. For example, byte references are allowed on byte boundaries, but 16-bit references are allowed only on 16-bit boundaries (address 0x0000, 0x0002, 0x0004, etc). 32-bit references are allowed only on 32-bit boundaries (addresses 0x0000, 0x0004, 0x0008, 0x000C, etc).

### 19.5.6 Errors on XBus

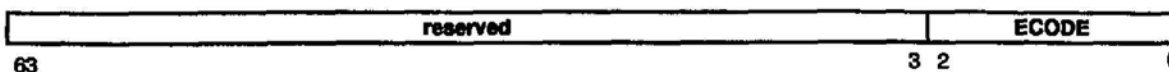
Errors on XBus can be reported in one of three ways. One way is in error reply packets that are reply packets with the E bit set. The second method of reporting errors is through the XPAR[3:0] signals. The third method is on the CCERR pin. See Section 16.8.

## Error Reply Packets

Reply packets may have the E bit set in the header to indicate that the corresponding request could not be processed correctly. The second cycle of an error reply provides additional error information, and the remaining cycles, if it is a nine cycle reply, are ignored. Note that the length of an error reply packet is the same as the length of the corresponding normal reply packet.

The format of the Error Data Cycle is shown in Figure 19-7. The Error Data Cycle has the normal parity for data cycles, all odd.

Figure 19-7. Format of Error Data Cycle



The least significant three bits of the error data cycle convey an error code, ECODE. Table 19-3 shows the encoding of the ECODE field.

Table 19-3. Encoding of the ECODE Field of the Error Data Cycle

ECODE	Meaning
000	reserved
001	UC: Uncorrectable Error
010	TO: Timeout
011	BE: Bus Error
100	UD: Undefined Error
101	reserved
110	reserved
111	reserved

The ECODE field encodes errors. The meanings of the codes are largely uninterpreted and are logged in the MXCC's error register (see MXCC Error Register, page 0-42). The meanings of the codes are similar to those of the MBus errors of the same name.

☐ **BE (Bus Error)**

This error code indicates a bus error such as a parity error on the system bus. It can also be used to indicate another system implementation dependent error.

☐ **TO (timeout)**

This code may be generated by a BW after some length of time has elapsed without a reply from the system bus. This error code can also be used to indicate a system implementation-dependent error.

Note that XBus does not have transaction timeouts as such. But BWs may generate timeout errors for transactions that have taken longer than expected to complete. Since no device on XBus will take responsibility for them, packets addressed to an invalid XDST will be lost and never generate a timeout error.

☐ UC (uncorrectable)

This code is signalled when the addressed system device (usually a memory controller) encounters an uncorrectable error (such as parity, uncorrectable ECC, etc) in accessing the data. This error code can also be used to indicate a system implementation-dependent error.

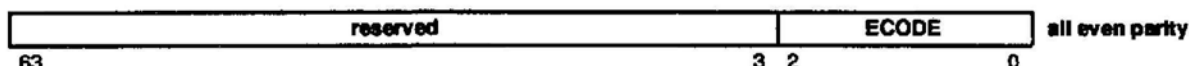
☐ UD (undefined)

This code indicates a system implementation-dependent error.

### Memory Fault Cycle

Parity may be used to communicate a data error. The sending device can drive all-even parity on a data cycle. This indicates that a memory fault condition exists for the corresponding data. A data cycle with all even parity is called a memory fault cycle. The format of a memory fault cycle is shown in Figure 19-8.

Figure 19-8. Format of Memory Fault Cycle



The least significant three bits of the error data cycle conveys an error code, ECODE. Figure 19-3 shows the encoding of the ECODE field.

### Transmission Parity Errors

A parity error can happen at any time. Parity that is neither all even nor all odd indicates an error in the transmission on XBus. The MXCC reports an XBus transmission parity error by asserting the  $\overline{CCERR}$  pin.

### E-cache Parity Errors

When the MXCC detects a parity error in the external cache, the MXCC asserts the  $\overline{CCERR}$  pin. MXCC and the SSP both generate and check even parity on VBus when VBus parity checking is enabled in  $\overline{CCCR}.PE$  and  $\overline{MCNTL}.PE$ , respectively.

## 19.6 Cache-Consistency Protocols

The MXCC supports full cache consistency on the system bus and between the SSP and the MMC. MBus and XBus configurations differ in consistency support and protocol. In the MBus configuration, the MXCC supports an MBus cache-consistency protocol. In the XBus configuration, the cache-consistency protocol is supported cooperatively by the MXCC and the bus watchers.

Both protocols implement data ownership. A sub-block of data may be owned by a particular cache in the system. This owner is responsible for supplying the data and for writing the data back to main memory. Memory owns all data not owned by any cache.

### 19.6.1 Sub-Block States

The E-cache tags store cache-consistency states for each sub-block. Three bits are used to encode the state, but only five states are used. The encoding of the consistency states is shown in Table 19-4.

Table 19-4. Encoding of Cache-Consistency States

STATES	S	O	V
Invalid (INV)	X	X	0
Exclusive & Clean (EX&C)	0	0	1
Exclusive & Dirty (EX&D)	0	1	1
Shared & Clean (SH&C)	1	0	1
Shared & Dirty (SH&D)	1	1	1

In addition, there is another bit per sub-block in the E-cache tags—the P bit—that is used only to prevent the SSP from sending multiple accesses to the same sub-block. This bit does not participate in the cache-consistency protocols.

### 19.6.2 State Transitions

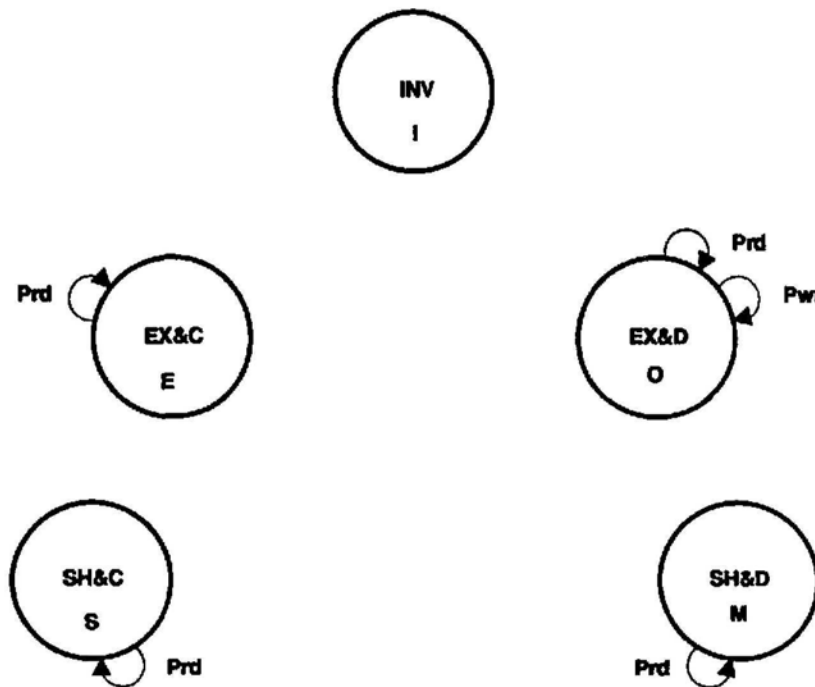
In the XBus configuration, the MXCC implements only a very few state transitions directly. Instead, the BWs direct the state transitions on all accesses from the system bus and on any accesses by the SSP that require access to the system bus. Only transitions for processor read hit and processor exclusive write ( $S=0, O=1$ ) are implemented in the MXCC.

The BWs control the cache-consistency state by passing a state transition command using the TCMD field of the XBus command header. The TCMD field indicates how the S, O, and V bits should be updated for the transaction. This scheme allows the BW to implement the state transition diagram of the system bus to which it is connected without using a fixed algorithm. A BW will command transactions that are not shown in Figure 19-9 in order to implement the system bus's cache-consistency scheme.

The owner of a sub-block is the E-cache of the processor that last wrote to the sub-block. When the SSP issues a write to a sub-block not exclusively owned ( $S=1$  or  $O=0$ ), the MXCC issues a write command to the BW. The BW communicates the shared write to other caches in the system on the system bus. The BW can then send a write reply to the MXCC with the TCMD field, indicating that the MXCC is to assume ownership.

The state transitions implemented directly in the MXCC are shown in Figure 19-9.

Figure 19-9. MXCC XBus Cache-Consistency State Diagram



## 19.7 XBus Commands

Bus commands are carried in the header cycle of each packet. XBus defines two types of commands:

- ☐ Data commands (DCmd).
- ☐ Tag commands (TCmd).

### 19.7.1 Data Commands

The data command is used by the sender to get data from or put data into another device. The data command is composed of the PTYP and RPLY fields of the header. When RPLY is 0, the command is a request. When it is 1, the command is a reply. Table 19–5 shows the data commands by PTYP and the packet length of the request and reply packets of each PTYP.

Both types of command are present in the header cycle of a packet. The two commands operate independently.

Table 19–5. Data Commands

Command	PTYP	Length	
		Request	Reply
NoOp	00000	2	2
NC Get Block	00010	2	9
Flush Block	00011	2	9
Get Single	00100	2	2
Put Single	00101	2	2
Get Block	00110	2	9
Put Block	00111	9	2 or 9
I/O Get Single	01000	2	2
I/O Put Single	01001	2	2
I/O Get block	01010	2	9
I/O Put block	01011	9	2
Demap	01110	2	2
Interrupt	01111	2	2
Swap single	10101	2	2
I/O Swap single	11001	2	2



### **NoOp Request**

The NoOp command indicates that no data is being transferred. It is useful because it allows the tags of an entry to be manipulated via TCMD without having to transfer data.

The NoOp Request packet is two cycles long. The first cycle is the header; the second cycle is unused.

### **NoOp Reply**

A NoOp Reply packet is used to reply to a NoOp Request packet. It indicates acknowledgement of the NoOp Request packet. PTYP, SIZE and ADDRESS are identical to those in the request header, and the XDST field is identical to the request packet's XSRC field.

### **Non-Cacheable Get Block Request**

NC Get Block is used by an XBus device to read a block from physical address space (memory space as opposed to I/O space) when it does not intend to cache the block.

An NC Get Block Request packet requests that the block specified by the ADDRESS field in the header be returned via an NC Get Block Reply. The packet is two cycles long. The first cycle contains the header; the second cycle is unused.

### **Non-Cacheable Get Block Reply**

An NC Get Block Reply packet is a response to an earlier NC Get Block Request. The packet is nine cycles long. The header cycle reflects most of the information in the request header, while the eight cycles of data supply the requested block in *critical word first* order. Critical word first order means that the byte addressed in the request is returned in the first data cycle, and additional double-words of the block at increasing addressed modulo 64 (the block size in bytes) are sent in successive data cycles.

PTYP, SIZE, and ADDRESS are identical to those in the request header; the XDST field is identical to the request packet's XSRC field.

### **Flush Block Request**

This command is neither sent nor handled by the MXCC.

### ***Flush Block Reply***

The MXCC generates Flush Block Reply packets automatically after it generates a Get Block Request packet. These Flush Block Reply packets have no corresponding request packet. They are used by the MXCC to return dirty sub-blocks from the victimized block to memory. The victimized block is the block that will contain the newly read data from the Get Block Reply packet.

The packet is nine cycles long. The eight cycles of data transmit the data in natural order. Natural order means that bytes 0–7 of the cache block are in the first data cycle, followed by data for increasing addresses in successive cycles. ADDRESS[5:0] are zero. SIZE is 64 bytes, and ADDRESS[35:8] is the physical address of the block E-cache. The XDST field is 0x10. ADDRESS[7:6] gives the number of the sub-block within the block being flushed.

### ***Get Single Request***

This command is used to fetch a single (up to doubleword) datum from the 36-bit physical address space. It is not supported or generated by the MXCC.

### ***Get Single Reply***

A Get Single Reply packet is two cycles long. The header cycle reflects most of the information in the request header, while the data cycle supplies the requested single. PTYP, SIZE and ADDRESS are identical to those in the request header, and the XDST field is identical to the request packet's XSRC field.

### ***Put Single Request***

This command is used to store a single (up to 64 bits) datum into the 36-bit physical address space.

The Put Single Request packet requests that a given value be written into the single specified by the ADDRESS and SIZE fields of the header. Note that the value to be written may be a byte, a half-word, a word, or a doubleword as specified in the SIZE field. A Put Single packet is two cycles long. The first cycle is the header; the second cycle contains the value to be written.

### ***Put Single Reply***

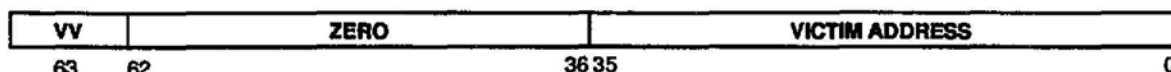
A Put Single Reply packet is used to acknowledge an earlier Put Single Request. The header cycle reflects most of the information from the request packet header; the second cycle contains the data that was written, just as in the request packet. PTYP, SIZE, and ADDRESS are identical to those in the request header, and the XDST field is identical to the request packet's XSRC field.

## Get Block Request

This command is used to read a block from physical address space with the intent of caching it.

A Get Block Request packet requests that the block specified by the ADDRESS field in the header cycle be returned via a Get Block Reply packet. A Get Block Request packet is two cycles long. ADDRESS[2:0] are ignored by the MXCC and always zero in Get Block Request packets issued by the MXCC. The first cycle is the header and the second cycle contains the address of the block being victimized, if any. The format of the victim address is shown in Figure 19-10.

Figure 19-10. Format of Victim Address Data Cycle



The fields of the victim address data cycle are:

- VV**                      Victim Valid:
- 0: Invalid. No victim.
  - 1: Valid victim address.
- ZERO**                      Always zero.

**VICTIM ADDRESS**      Physical address of the cache block to be replaced.

If VV is clear, there are no valid sub-blocks in the selected block to be replaced. If VV is set, there are one or more valid sub-blocks in the selected block to be replaced. The valid sub-blocks may be clean or dirty. If a victim sub-block is dirty, it will be written with a Flush Block Reply packet after the Get Block Request has been sent.

## Get Block Reply

A Get Block Reply packet is a response to an earlier Get Block Request. The packet is nine cycles long. The header cycle reflects most of the information in the request packet header, while the eight data cycles transmit the requested block of data in critical word first order (see Non-Cacheable Get Block Reply, page 19-24).

PTYP, SIZE and ADDRESS are identical to those in the request header, and the XDST field is identical to the request packet's XSRC field.

### ***Put Block Request***

This command is used to write a block into physical address space.

A Put Block Request packet requests that a given value be written into the block specified by the ADDRESS field of the header. A Put Block packet is nine cycles long. The first cycle is the header; the remaining eight cycles transmit the data to be written in natural order (see Flush Block Reply, page 19-25). ADDRESS[5:3] must be zero. ADDRESS[2:0] are ignored.

### ***Put Block Reply***

A Put Block Reply packet is a response to an earlier Put Block Request packet. There are two sizes of Put Block Reply packets, two cycles and nine cycles. A two-cycle Put Block Reply simply acknowledges that the Put Block Request has been performed. The first cycle is the header; the second cycle is unused.

A nine-cycle Put Block Reply is used for MXCC stream operations. It not only acknowledges the Put Block Request, but also supplies the data to be written. The eight data cycles are in natural order.

In both sizes of Put Block Reply packets, the PTYP, SIZE, and ADDRESS fields are identical to those in the request packet header, and the XDST field is identical to the request packet's XSRC field. ADDRESS[5:0] are ignored and must be zero.

The stream hardware in the MXCC can cause a write of data cached in the E-cache RAM. The BW is responsible for snooping all of its own operations, including stream writes. When a BW detects a tag match on a stream write, the BW returns a nine-cycle stream write reply containing the 64 bytes of data. The MXCC writes the data into the E-cache RAM and invalidates the internal caches in the SSP by issuing invalidate cycles on the VBus.

### ***IO Get Single Request***

This command is used to fetch a single (up to doubleword) datum from the IO address space.

An IO Get Single Request packet requests that the single datum specified by ADDRESS and SIZE fields in the header cycle be returned via an IO Get Single Reply packet. An IO Get Single Request packet is two cycles long. The first cycle is the header; the second cycle is unused.

### ***IO Get Single Reply***

An IO Get Single Reply packet is a response to an earlier IO Get Single Request packet. This packet is two cycles long. The header cycle reflects most of the information in the request header, while the data cycle supplies the requested single. PTYP, SIZE and ADDRESS are identical to those in the request header, and the XDST field is identical to the request packet's XSRC field. The data cycle contains the data. For SIZE less than 64 bits, the requested data is in its natural position based on ADDRESS and SIZE. For example, a single byte requested from an address with ADDRESS[2:0] = 6 would be on DATA[15:8]. Other fields of DATA should be ignored by the recipient of the reply.

### ***IO Put Single Request***

This command is used to write a single (up to 64 bits) datum into the IO address space.

The IO Put Single Request packet requests that a given value be written into the single specified by the ADDRESS and SIZE fields of the header. The value to be written may be a byte, a half-word, a word or a doubleword as specified in the SIZE field. An IO Put Single packet is two cycles long. The first cycle is the header, while the second cycle contains the value to be written. The data sent in the data cycle must be aligned into the field of DATA[63:0] that corresponds to the selected ADDRESS and SIZE. For example, a single byte to be stored in an address with ADDRESS[2:0] = 0 must be supplied on DATA[63:56]. Other fields of DATA are ignored by the recipient.

### ***IO Put Single Reply***

An IO Put Single Reply packet acknowledges that an earlier IO Put Single Request is complete. The header cycle reflects most of the information from the request packet header; the second cycle is unused. PTYP, SIZE, and ADDRESS are identical to those in the request header, and the XDST field is identical to the request packet's XSRC field.

### ***IO Get Block Request***

This command is used to read a block of data from the IO address space.

An IO Get Block Request packet requests that the block specified by the ADDRESS field in the header be returned via an IO Get Block Reply packet. An IO Get Block Request packet is two cycles. The first cycle contains the header. The second cycle is unused. ADDRESS[5:3] must be zero. ADDRESS[2:0] are ignored.

### ***IO Get Block Reply***

An IO Get Block Reply packet is a response to an earlier IO Get Block Request packet. The packet is nine cycles long. The header cycle reflects most of the information in the request header, while the eight data cycles transmit the request block of data in natural order (see Flush Block Reply, page 19-25).

PTYP, SIZE, and ADDRESS are identical to those in the request header. The XDST field is identical to the request packet's XSRC field.

### ***IO Put Block Request***

This command is used to write a block of data into the IO address space.

An IO Put Block packet requests that a given value be written into the block specified by the ADDRESS field of the header. The packet is nine cycles long. The first cycle contains the header; the subsequent cycles contain the data. The data is present in natural order (see Flush Block Reply, page 19-25). ADDRESS[5:3] must be zero. ADDRESS[2:0] are ignored.

### ***IO Put Block Reply***

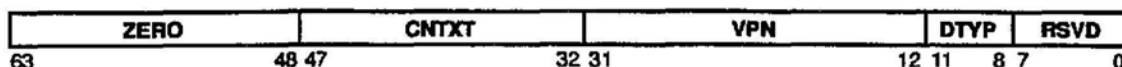
An IO Put Block Reply packet acknowledges that the write requested by an earlier IO Put Block Request is complete. The packet is two cycles long. The first cycle contains the header; the second cycle is unused. PTYP, SIZE, and ADDRESS are identical to those in the request packet. The XDST field is identical to the request packet's XSRC field.

### ***Demap Request***

Demap is used to remove the virtual to physical mapping for one or more virtual pages. When a processor demaps a page, its MXCC issues a Demap Request on XBus with XDST = 0x10 (BW). A BW communicates the request over a system bus. Other BWs on the system bus react to the request by sending a Demap Request packet on their local XBuses to their MXCCs. Once an MXCC has acted on the Demap Request from the bus, it replies with a Demap Reply packet to its local BWs. When all processor/MXCC pairs have replied, the originating BW replies to its MXCC with a Demap Reply packet, indicating that the Demap has been completed by all processors.

A Demap Request packet is two cycles. This first cycle is the header cycle. The ADDRESS field of the header must be zero. The second cycle of the packet is the demap data cycle. The format of the demap data cycle is shown in Figure 19-11.

Figure 19–11. Demap Data Cycle Format



ZERO	Unused 16-bit field of zeros.
CNTXT	Context.
VPN	Virtual Page Number.
DTYP	Demap Type. The encoding of this field is identical to the SuperSPARC processor demap types in Table 9–6.
RSVD	Reserved. Must be zero.

### Demap Reply

A Demap Reply packet acknowledges an earlier Demap Request packet. The reply is two cycles. The first cycle contains the header. The ADDRESS field in a Demap Reply must be zero. The second cycle is unused.

### Interrupt Request and Interrupt Reply

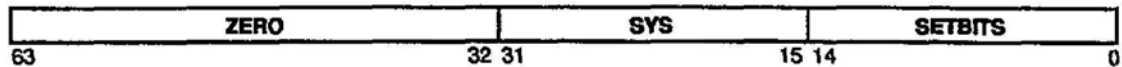
This command is used to generate processor interrupts. Each interrupt has an initiating XBus, and one or more target XBuses. The initiating XBus contains the device that generated the interrupt, while the target XBuses are the ones whose processors are to be interrupted. The interrupt command is used both at the initiating end and the target ends.

When a processor initiates an interrupt, its MXCC sends an Interrupt Request packet to its BWs, one of which communicates the request on a system bus. The BW also sends an Interrupt Reply packet to the initiating MXCC when the Interrupt Request has been successfully sent on a system bus. This reply to the originating MXCC serves as an acknowledgement. If the originating MXCC is also one of the targets of the Interrupt Request, the reply packet also serves to request that MXCC interrupt the local processor.

When a processor is the target of an interrupt initiated from another XBus, the interrupt request arrives over the system bus, and the target BW sends the request to the target MXCC as an Interrupt Request packet. Once the target MXCC has interrupted its processor, it acknowledges the Interrupt Request packet with an Interrupt Reply packet.

The Interrupt Request packet is a two-cycle packet. The first cycle contains the header. The ADDRESS field of the header must be zero. The second cycle is an Interrupt Data Cycle. The format of an Interrupt Data Cycle is shown in Figure 19–12.

Figure 19–12. Interrupt Data Cycle Format



ZERO	Unused 32-bit field of zeros.
SYS	Uninterpreted information passed through from MXCC interrupt generation register to the BW for system-specific use.
SETBITS	Bits to Set. Specifies the bits that are to be set in the target MXCC's interrupt pending register. See Section 16.7.

### Swap Single Request

This command is used to perform an atomic load-store on a single datum (up to 64 bits) in the 36-bit physical address space.

A Swap Single Request packet requests that a given value be atomically written into the single specified by the ADDRESS and SIZE fields of the header. The value written may be a byte, half-word, word, or doubleword, as specified by the SIZE field. A Swap Single Request packet is two cycles long. The first cycle is the header; the second cycle contains the value to be written.

As in Put Single Request and Get Single Reply, if the SIZE is less than 64 bits, the data is on the field of DATA determined by SIZE and ADDRESS[2:0]. Other fields of DATA are ignored and may contain unrelated information.

### Swap Single Reply

A Swap Single Reply packet is used to acknowledge an earlier Swap Single Request. The header cycle reflects most of the information in the request header, while the second cycle contains the data to be written just as in the request packet. This is the same as for a Put Single Reply because the swap operation is actually performed in cache, not in the system memory.

PTYP, SIZE, and ADDRESS are identical to those in the request packet header. The XDST field is identical to the request packet's XSRC field.

### IO Swap Single Request

This command performs an atomic load-store a single datum (up to 64 bits) into the IO space.

An IO Swap Single Request packet requests that a given value be written into the location specified by ADDRESS and SIZE fields of the header and the old value be returned. The value may be a byte, half-word, word or doubleword as specified by the SIZE field. The packet is two cycles long. The first cycle is the header; the second cycle is the data value to be written.



### IO Swap Single Reply

An IO Swap Single Reply packet acknowledges that an earlier IO Swap Single Request is complete. The first cycle contains the header; the second cycle contains the data returned by the destination device for the atomic load-store. The returned value is presumably the location's previous contents.

PTYP, SIZE, and ADDRESS are identical to those in the request packet header. The XDST field is identical to the request packet's XSRC field.

### 19.7.2 Tag Commands

Tag commands are used to keep cache tags in the MXCC for E-cache and the tags in the SSP consistent with system bus tags in the BW(s). The six-bit TCMD field within a packet header specifies how the tag for the sub-block referenced by the address part of the header should be manipulated. BWs and the MXCC use these TCMDs to keep each other's tags synchronized. The portion of the tag of concern to tag commands contains some address comparison (ACOMP) bits and four flag fields:

☐ Shared

The shared bit indicates whether the entry is present in more than one cache on the system bus.

☐ Owner

The owner bit indicates whether this copy should respond to reads over the system bus.

☐ Valid

The Valid field indicates whether this entry is valid. The tag entry is valid if this field is one and invalid if the field is zero.

☐ Pending

The pending bit is useful for packet switched busses indicating whether an entry has a packet outstanding on the system bus. A packet is outstanding if a request packet has been sent for which no reply has been received.

The ACOMP field of the tags contains those bits of ADDRESS that are not used to index into the tag array.

The TCMD field of a packet header consists of three sub-fields, as shown in Figure 19-13. TCMD encodes the tag command and applies to the sub-block addressed by the address field of the header.

Figure 19-13. TCMD Field of Header Cycle



The sub-fields of TCMD are SHCMD, OWCMD, and VCMD. Their actions are largely but not entirely rotational. When SHCMD and OWCMD are both 11, neither of these fields has any effect. The interpretation of the sub-fields is as follows:

**SHCMD:** Shared Bit Command. Indicates how the shared bit for the addressed sub-block should be manipulated. See Table 19-6.

*Table 19-6. Shared Bit Commands*

SHCMD	ACTION
00	Write a value of 0
01	Write a value of 1
10	Expect a value of 0
11	Expect a value of 1 or No-op when OWCMD is also 11

The two codes for expected values are provided to the destination device for information and error checking. The MXCC reports a cache-consistency error (see Section 15.8) if the expected value from the SHCMD field of a packet from the system does not match the E-cache tag S bit of the addressed sub-block. A BW implementation is not required to check that the local copy of the tag S bit has the expected value, but may do so for better error detection.

**OWCMD:** Owner Bit Command. Indicates how the owner bit should be manipulated. See Table 19-7.

*Table 19-7. Owner Bit Commands*

OWCMD	ACTION
00	Write a value of 0
01	Write a value of 1
10	Expect a value of 0
11	Expect a value of 1 or No-op when SHCMD is also 11

The two codes for expected values are provided to the destination device for information and error-checking. The MXCC reports a cache-consistency error (see Section 15.8) if the expected value from the OWCMD field of a packet from the system does not match the E-cache tag's O bit for the addressed sub-block. A BW implementation is not required to check that the local copy of the tag O bit has the expected value, but may do so for better error detection.

VCMD:

Valid Bit Command. Indicates how the valid bit should be manipulated. See Table 19-8.

Table 19-8. Valid Bit Commands

VCMD	ACTION
00	Ext Write Invalidate
01	Clear Valid
10	Set Valid and Write Tag
11	NoOp

Ext Write Invalidate and Clear Valid are the same to the MXCC. Ext Write Invalidate is used by a BW to signal the MXCC that the valid bits for a cache sub-block should be cleared as a result of an external write from another processor. Clear Valid is used to remove a block from the cache. The action of these VCMDs is conditional on the P (pending) bit of MXCC's sub-block tag. If the P bit is clear (no operation pending on this sub-block), the valid bit for the sub-block is cleared. If the P bit is set, another operation is pending on this sub-block, and the VCMD is ignored.

Set Valid and Write Tag is used by a BW to signal MXCC that the information is new and the processor-side tags should be updated to maintain consistency between the tags.

## 19.8 MXCC Use of XBus

The MXCC issues only packets with certain combinations of header fields. Furthermore, it expects only certain packets from XBus. Other combinations are not used. BW designers will find the following information helpful in designing BWs to work with the MXCC.

### 19.8.1 XBus to MXCC

Table 19-9 contains the header field bits that the MXCC expects in request packets from XBus. The patterns in the TCmd field may differ from those shown, and any valid TCmd may be used. Different TCmd fields may be needed to support different system buses.

The MXCC Op field contains the name of the internal operation in the MXCC that generates this packet for packets from an MXCC or the internal operation generated by this packet for packets to an MXCC. In the MXCC Op field, NC means "Non-Cacheable," and CD means "Cache Disabled".

Table 19-9. Summary of XBus Request Packets to MXCC

MXCC OP	PTYP	R	L	E	XSRC	XDST	SIZE	TCMD	XBus Request
BUS READ BLOCK	00110	0	0	0	100nn	00000	101	011111	Get Block
BUS DEMAP	01110	0	0	0	100nn	00000	000	111111	Demap
BUS INTERRUPT	01111	0	0	0	100nn	00000	000	111111	Interrupt
BUS NC READ	01000	0	0	0	100nn	00000	0ab	111111	IO Get Single
BUS NC WRITE	01001	0	0	0	100nn	00000	0ab	111111	IO Put Single
BUS STAT UPDATE	00000	0	0	0	100nn	00000	000	xxxxxx	No Op
BUS LDST	10101	0	0	0	100nn	00000	0ab	110011	Swap Single
BUS WRITE INV	00101	0	0	0	100nn	00000	0ab	11000x	Put Single
BUS WRITE	00101	0	0	0	100nn	00000	0ab	110011	Put Single
BUS STREAM WR	00111	0	1	0	100nn	00000	101	1x0011	Put Block
BUS STREAM READ	00010	0	0	0	100nn	00000	101	1x1111	NC Get Block
BUS READ BLOCK	00110	0	0	0	100nn	00000	101	011111	Get Block

In Table 19-9, values are indicated in binary. The following symbols are used to indicate certain variable data:

- x Packet-dependent. Can be either 0 or 1.
- E This bit is set on a reply when the corresponding request cannot be processed correctly. When the error bit is set, the least significant three bits of the first data cycle provide additional information about the error.

ab	Data Size. The data size is encoded into these two bits as: <ul style="list-style-type: none"><li>■ 00: Byte</li><li>■ 01: Halfword</li><li>■ 10: Word</li><li>■ 11: Doubleword</li></ul>
nn	BW number may be here. If there is only one BW, it is BW number 01. If there are only two BWs, they are BW numbers 00 and 01.

### ***Summary of XBus Reply Packets***

Table 19–10 contains the header field bits that the MXCC expects in reply packets from XBus. Reply packets always match a previous request packet sent by the MXCC.

The patterns in the TCmd field may differ from those shown. Any valid TCmd may be used. Different TCmd fields may be needed to support different system buses. The ERR field is normally 0 but may be 1 if an error was encountered when the request was processed.

Table 19–10. Summary of XBus Reply Packets to MXCC

MXCC OP	PTYP	R P L Y	P L E N	E R R	XSRC	XDST	SIZE	TCMD	XBus Reply
CD LDST REPLY	10101	1	0	E	10000	01011	0ab	0x0010	Swap Single
CD WRITE REPLY	00101	1	0	E	10000	01010	0ab	0x0010	Put Single
CD RD BLK REPLY	00110	1	1	E	10000	01100	101	0x0010	Get Block
CD READ REPLY	00110	1	1	E	10000	01101	101	0x0010	Get Block
NC STRM WR REPLY	01011	1	0	E	10000	00000	101	111111	IO Put Block
NC STRM RD REPLY	01010	1	1	E	10000	00000	101	111111	IO Get Block
NC LDST REPLY	11001	1	0	E	10000	00001	0ab	111111	IO Swap Single
NC WRITE REPLY	01001	1	0	E	10000	00000	0ab	111111	IO Put Single
NC READ REPLY	01000	1	0	E	10000	00000	0ab	111111	IO Get Single
PREFETCH REPLY	00110	1	1	E	10000	00011	101	0x0010	Get Block
STREAM WR REPLY	00111	1	X	E	10000	00000	101	111111	Put Block
STREAM RD REPLY	00010	1	1	E	10000	00000	101	111111	NC Get Block
DEMAP REPLY	01110	1	0	E	10000	00000	000	111111	Demap
SHARED LDST RPLY	10101	1	0	E	10000	00001	0ab	0x0110	Swap Single
SHARED WR REPLY	00101	1	0	E	10000	00000	0ab	0x0110	Put Single
WR MISS REPLY	00110	1	1	E	10000	00000	101	0x0x10	Get Block
READ MISS REPLY	00110	1	1	E	10000	00001	101	0x0010	Get Block
BURST READ REPLY	00110	1	1	E	10000	00010	101	0x0010	Get Block
INT REPLY BW	01111	1	0	E	10000	00001	000	111111	Interrupt
INT RPLY CC	01111	1	0	E	10000	00000	000	111111	Interrupt

### 19.8.2 MXCC to XBus

Table 19–11 contains the header field bits that the MXCC sends in request packets to XBus.

Table 19–11. XBus Requests from MXCC to XBus

MXCC OP	PTYP	R P L Y	P L E N	XSRC	XDST	SIZE	TCMD	XBus Request
BURST READ MISS	00110	0	0	00010	10000	101	111111	Get Block
READ MISS	00110	0	0	00001	10000	101	111111	Get Block
WRITE MISS	00110	0	0	00000	10000	101	111111	Get Block
SHARED WRITE	00101	0	0	00000	10000	0ab	1x1x11	Put Single
SHARED WRITE INV	00101	0	0	00000	10000	0ab	1x1x01	Put Single
SHARED LDST	10101	0	0	00001	10000	0ab	1x1x11	Swap Single
DEMAP	01110	0	0	00000	10000	000	111111	Demap
STREAM READ	00010	0	0	00000	10000	101	111111	NC Get Block
STREAM WRITE	00111	0	1	00000	10000	101	111111	Put Block
PREFETCH	00110	0	0	00011	10000	101	111111	Get Block
NC READ	01000	0	0	00000	10000	0ab	111111	IO Get Single
NC WRITE	01001	0	0	00000	10000	0ab	111111	IO Put Single
NC STREAM READ	01010	0	0	00000	10000	101	111111	IO Get Block
NC STREAM WRITE	01011	0	1	00000	10000	101	111111	IO Put Block
NC LDST	11001	0	0	00001	10000	0ab	111111	IO Swap Single
CD READ	00110	0	0	01101	10000	101	111111	Get Block
CD READ BLOCK	00110	0	0	01100	10000	101	111111	Get Block
CD WRITE	00101	0	0	01010	10000	0ab	111111	Put Single
CD LDST	10101	0	0	01011	10000	0ab	111111	Swap Single
INTERRUPT	01111	0	0	00001	10000	000	111111	Interrupt

In Table 19–11, values are indicated in binary. The following symbols are used to indicate certain variable data:

- x** Packet-dependent. May be either 0 or 1.
- ab** Data Size. The data size is encoded into these two bits as:
- 00: Byte
  - 01: Halfword
  - 10: Word
  - 11: Doubleword

### ***Replies from MXCC to XBus***

Table 19–12 contains the header field bits that the XBus expects in reply packets from MXCC. The patterns in the TCmd field may differ from those shown. Any valid TCmd may be used. Different TCMD fields may be needed to support different system buses.

***Table 19–12. Replies from MXCC to XBus***

<b>MXCC op</b>	<b>PTYP</b>	<b>R P L Y</b>	<b>P L E N</b>	<b>XSRC</b>	<b>XDST</b>	<b>SIZE</b>	<b>TCMD</b>	<b>XBus Reply</b>
INVALIDATE REPLY	00000	1	0	00000	10000	0xx	111101	NoOp
READ BLOCK REPLY	00110	1	1	00000	10000	101	1x1x11	Get Block Reply
STREAM RD REPLY	00010	1	0	00000	10000	101	1x1x11	NC Get Block Reply
STREAM WR REPLY	00111	1	0	00000	10000	101	111111	Put Block Reply
NC READ REPLY	01000	1	0	00000	10000	0ab	111111	IO Get Single Reply
NC WRITE REPLY	01001	1	0	00000	10000	0ab	111111	IO Put Single Reply
FLUSH REPLY	00011	1	1	00000	10000	101	1x1111	Flush Block Reply
DEMAP REPLY	01110	1	1	00000	10000	000	111111	Demap Reply



## **19.9 Write Hits in Stream Transfers**

The stream hardware in the MXCC can cause a write of data cached in the E-cache RAM. The bus watcher is responsible for snooping all of its own operations, including stream writes. When a BW detects a tag match on a stream write, the BW returns a nine-cycle stream write reply containing the 64 bytes of data. MXCC writes the data into the E-cache RAM and invalidates the internal caches in the SSP by issuing invalidates on the VBus.

## 19.10 Arbitration and Flow Control

The MXCC contains a built-in, pipelined arbiter to control access to XBus. The arbiter performs the basic function of controlling access to the bus so that only a single device drives XBus on each cycle. It contains additional functions to reduce the average time needed for arbitration.

The use of queues is inherent in packet-based communication. Queues have finite size, however, so too many packets destined for a single destination can exceed the capacity of a receiving device's queue. Flow control is the stopping of senders that may overflow a receiver's queue until it has sufficient capacity for the incoming packet.

Each request to the arbiter for access to XBus encodes the length and priority of the packet to be sent. Refer to Subsection 19.4.4 for information about the priority of XBus packets.

Arbitration and flow control are intimately related for XBus, and are both linked to packet priority. Each device has low- and high-priority queues. A packet of a certain length and priority can be sent if the receiving device has space for at least that length in the incoming queue for that priority. Otherwise, the sender is blocked until there is sufficient room in the receiving queue.

### 19.10.1 XREQ Decoding

BWs request use of XBus on two dedicated lines to the MXCC, which contains the XBus arbiter. The meaning of the  $XREQ_n[1:0]$  signals depends on the sequence of values on the two lines. The sequences used and their meanings are described in Table 19-13.

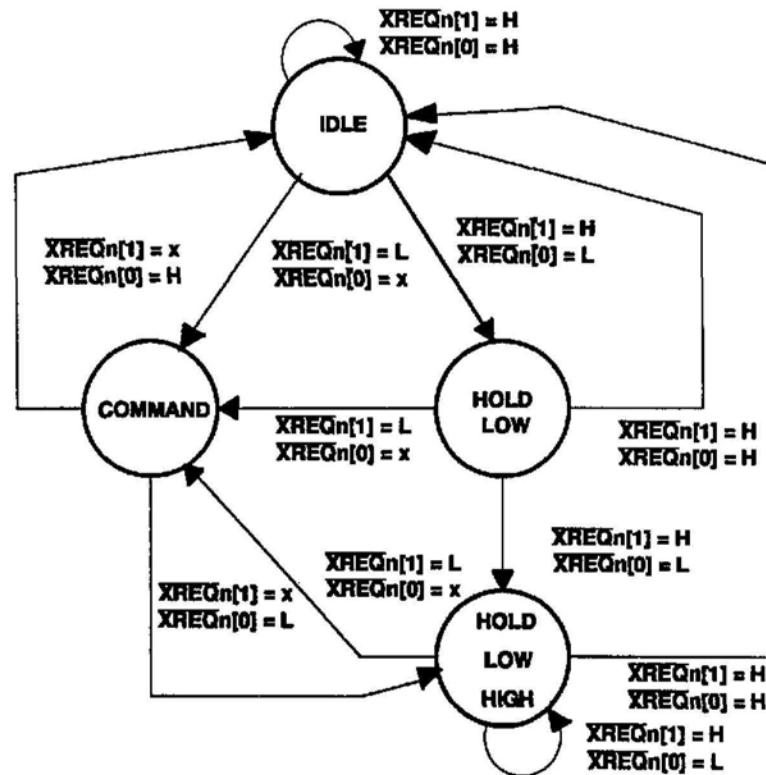
Table 19–13. Arbitration and Flow Control Encoding

FIRST CYCLE XREQn[1:0]	SECOND CYCLE XREQn[1:0]	MEANING
HH	—	Idle
HL		Block MXCC request packets for nine cycles
HL	HL	Block MXCC request and reply packets for nine cycles
LH	HH	Request XBus for low-priority two-cycle packet
LH	LH	Request XBus for low-priority nine-cycle packet
LH	HL	Request XBus for low-priority two-cycle packet and block MXCC packets for nine cycles
LH	LL	Request XBus for low-priority nine-cycle packet and block MXCC packets for nine cycles
LL	HH	Not valid
LL	LH	Request XBus for high-priority nine-cycle packet
LL	HL	Not valid
LL	LL	Request XBus for high-priority nine-cycle packet and block MXCC packets for nine cycles

The sequential nature of the XREQn[1:0] signals describes a finite state machine (FSM). Figure 19–14 represents the state machine that interprets the XREQ values to determine whether to prevent the MXCC from sending low- and high-priority messages.

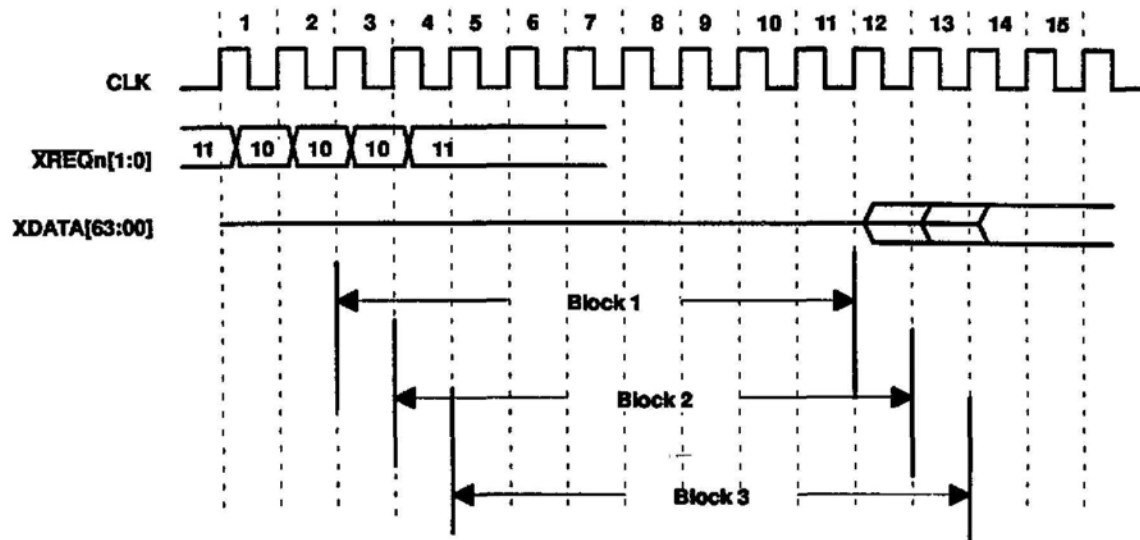
If a value of HH (i.e., neither XREQn[1] nor XREQn[0] asserted) has been present for two or more cycles, the state machine is at IDLE. If the machine receives a value of HL (XREQn[1] = false; XREQn[0] = true), the state machine moves to state HOLD LOW, the state where the low-priority queue is signalled to block for the next nine cycles. If, at the next clock, the value is again HL, the state machine moves to HOLD LOW HIGH, the state where both low- and high-priority queues are signalled to hold. If the value is HH, the state machine returns to IDLE. For any other value, the state machine goes to the command decode state COMMAND.

Figure 19–14. Message Blocking State Machine



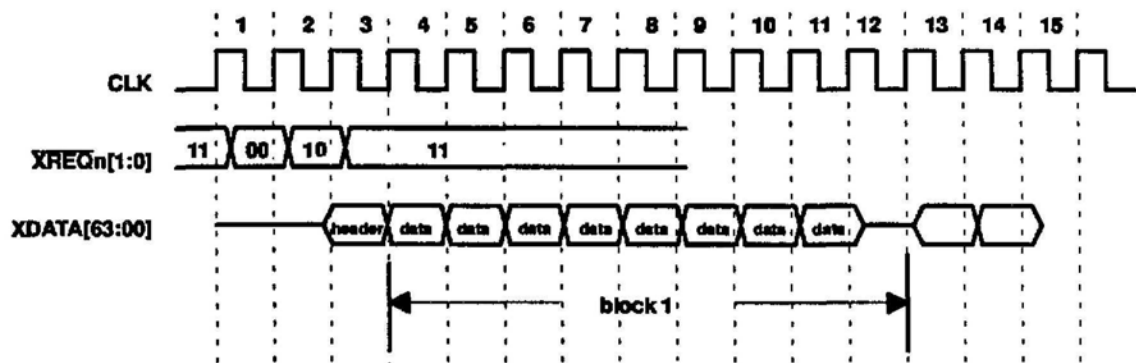
The  $XREQ_n$  encodings that block the REQUEST and REPLY packets from the MXCC are not cumulative. They signify that, from this time forward (for nine cycles), the MXCC should not send any REQUEST or REPLY packets. This is illustrated in Figure 19–15, which depicts message packets from the MXCC to a particular BW. In cycle 1 the BW inhibits REQUEST packets for nine cycles (i.e., the first REQUEST packet directed at this BW would be in cycle 12). In cycle 2 the BW inhibits all packets for nine cycles. (The earliest the MXCC could send a packet would be cycle 13.) In cycle 3 the BW again inhibits packets for nine cycles (now the earliest packet would be in cycle 14). Since  $XREQ_n$  signals are clocked before being used, the actual blocking is delayed for one cycle. The blocking period for the first block is from the start of cycle 3 to the end of cycle 11.

Figure 19–15. Sequential Blocking Of Messages



A header may arrive before the block becomes effective. Figure 19–16 shows a request for a low-priority packet and blocking of all MXCC packets. Note that a packet begins before the block occurs, and that the effective blocking period is for only one cycle.

Figure 19–16. Header Received Before Block



### 19.10.2 Arbitration Priorities

The XBus arbiter in the MXCC supports four priorities. Listed in descending priority order, they are:

CC HIGH

XBus arbitration requests from the MXCC to send reply packets to a BW (highest priority).

BW HIGH	XBus arbitration requests from BW to send block read reply packets to the MXCC.
BW LOW	XBus arbitration requests from BW to send system request packets and most system bus reply packets to the MXCC.
CC LOW	XBus arbitration requests from the MXCC to send request packets to a BW (lowest priority).

Servicing of requests at a specific level is round-robin. For example, all requests at BW LOW priority are granted before any XBus requests at CC LOW are granted.

When a high-priority reply packet is present in a BW (destined for the MXCC), the MXCC issues a grant for that high-priority packet, in effect changing the priorities for nine cycles to:

(BW HIGH > CC HIGH > BW LOW > CC LOW)

### 19.10.3 Flow Control in MXCC's Queues

There are six queues in the MXCC, as shown in Figure 19-5. The flow control mechanism for each is described below.

XIL/XIH	XIL holds low-priority packets from BWs, and XIH holds high-priority packets from BWs. They are flow-controlled by the arbiter. The arbiter refuses to grant requests to any of the BWs when either queue is above its high water mark.
ARBL	Holds arbitration requests from BWs to use XBus to send low-priority packets and is not flow-controlled. This FIFO contains as many entries as there are entries in the BW XIL FIFO. It cannot overflow because XIL does not overflow.
ARBH	Holds arbitration requests from BWs to use XBus to send REPLY packets. ARBH contains a single entry. It cannot overflow because the MXCC will permit only a single outstanding request packet that will generate high-priority reply packets from the BWs.
XOL/XOH	XIL holds low-priority packets to the BWs, and XIH holds high-priority packets to the BWs. These are prevented from overflowing because MXCC prevents the processor from accessing VBus when these queues reach their high water marks.

### 19.10.4 Flow Control in Bus Watcher's Queues

A prototypical BW has four queues. The flow control mechanism for each is described below.

BOL/BOH	BOL (BW outgoing low-priority requests) and BOH (BW outgoing high-priority replies) are flow-controlled by the BW via its $XREQn[1:0]$ lines, which can block either request or reply packets.
BIL/BIH	BIL (BW low-priority request replies) and BIH (BW high-priority read miss replies) must be flow-controlled on the system side of the BW. The BW must in some way prevent the system bus from issuing cycles that would result in reply or request packets being queued for the XBus when either of these queues is full.

### 19.10.5 Default Grantee

The minimum arbitration latency is five cycles if the requestor is a bus watcher and two cycles if the requestor is the MXCC. Therefore, assuming the XBus is idle, arbitration latency adds seven cycles to the cost of a cache miss.

The default grantee mechanism is a way to avoid this latency on most replies. When the MXCC issues a read miss to a system bus through a bus watcher, it gives grant to that BW in anticipation of the reply packet from the system bus. The BW experiences no arbitration delay when the reply arrives and can immediately transmit the reply packet on the XBus.

When a BW has default grant, it must be prepared for  $XGNTn$  to be deasserted by the MXCC at any time. If it has already begun to send a transaction, it must continue. The MXCC guarantees that no other device will obtain ownership of the XBus during the time the BW is sending packets.

MXCC selects which BW is the default grantee based on  $ADDRESS[9:8]$  and the number of BWs, according to Table 19-14.

Table 19-14. Default Grantee

Number of BWs	$ADDRESS[9:8] = 00$	$ADDRESS[9:8] = 01$	$ADDRESS[9:8] = 10$	$ADDRESS[9:8] = 11$
1	BW0	BW0	BW0	BW0
2	BW0	BW1	BW0	BW1
4	BW0	BW1	BW2	BW3

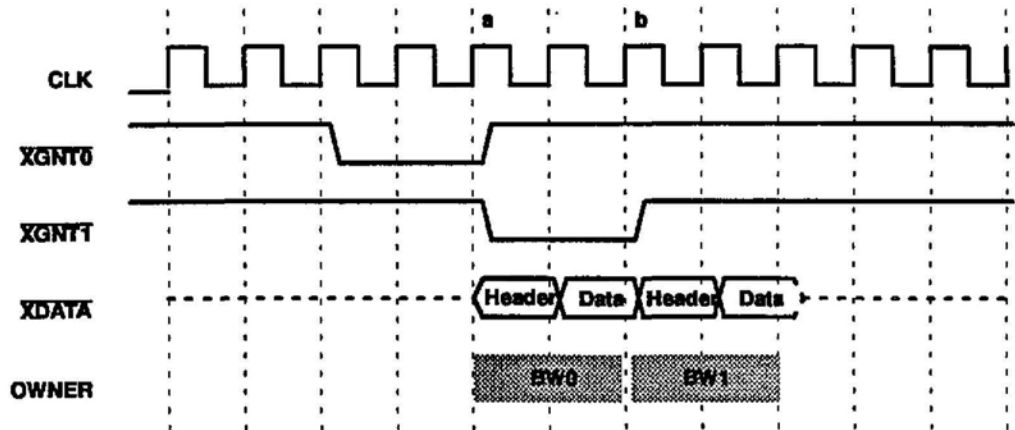
### 19.10.6 Bus Ownership

A device on XBus can only drive the bus when it owns the bus. Ownership is determined by arbitration.

### 19.10.6.1 No Default Grantee

The granted device owns the bus exactly two cycles after the grant has been issued (refer to Figure 19–17). The length of ownership is dependent on the length of the grant signal. The owner (with the exception of default grantee) must drive the XDATA and parity lines exactly two cycles after the grant has been issued. It must drive these lines only for the length of time XGNT<sub>n</sub> is asserted.

Figure 19–17. Bus Ownership



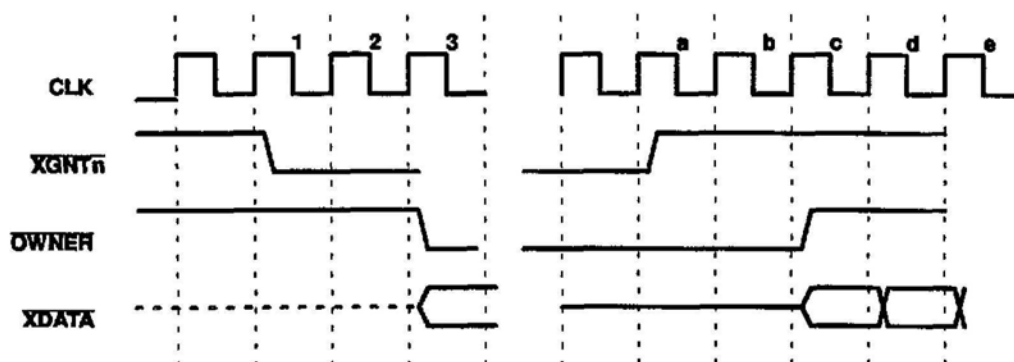
### 19.10.6.2 Default Grantee

A default grantee gains ownership of the bus exactly two cycles after the grant has been issued. BWs are given default grant only for nine-cycle packets. Under some circumstances the grant will be negated before the default grantee has used the bus. The change of ownership follows very exact rules:

- Rule 1: If the BW detects a grant and it did not issue a request for the bus (using the XREQ<sub>n</sub>[1:0] lines) two cycles previously, the grant must be a default grant.
- Rule 2: The BW may drive a high-priority reply message from the system bus any time a DEFAULT GRANT has been issued, and, once the message is started, it must finish driving the header and eight data words. Refer to Figure 19–18. The default grant (XGNT<sub>n</sub>) is issued in cycle 1 and latched into the BW at cycle 2. It controls BW internal logic during cycle 3. Cycle 3 is the earliest cycle in which a header may appear in reaction to the default grant. The MXCC removes the default grant at cycle *a*, and the last point at which the BW may drive a header is cycle *c*. The MXCC may issue a grant to another device during cycle *e* if no header was driven during cycle *c*.

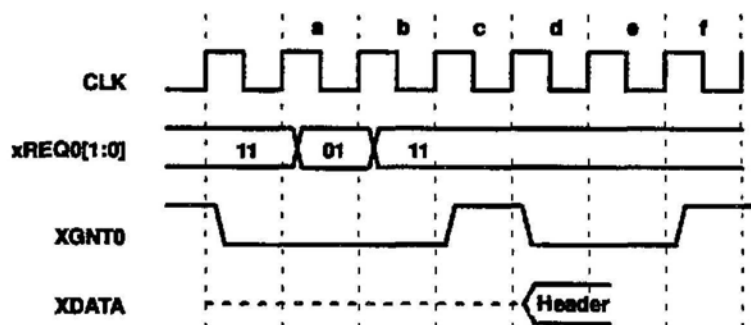


Figure 19–18. Default Grant Ownership



**Rule 3:** It is possible for a BW eligible to become a default bus grantee to issue a request for the XBus. Figure 19–19 shows such a situation. If the BW issues a request for the bus in cycle *a* and cycle *b*, the MXCC first reacts to the request in cycle *c* by removing the default bus grant. The last cycle in which the BW may issue a header for the default grant is cycle *d*. The grant from the MXCC for the low-priority request is issued during cycles *d* and *e*. The BW would drive the header during cycle *f*.

Figure 19–19. Request During Default Grant



### 19.10.7 Message Priority Detection

In order to correctly arbitrate for the XBus, BWs must be able to prioritize messages. Only system responses to GET BLOCK messages from the MXCC with XSRC ID of 0x02 or 0x01 should be considered high-priority messages.

## 19.11 XBus Cycle Waveforms

Figure 19–20 shows a simple two-cycle packet. The BW requests the use of the bus by asserting  $01_2$ , followed by  $11_2$ , on the  $XREQn[1:0]$  lines. The MXCC grants the BW the bus for two cycles, and the BW sends a two-cycle packet.

Figure 19–20. XBus Two-Cycle Packet

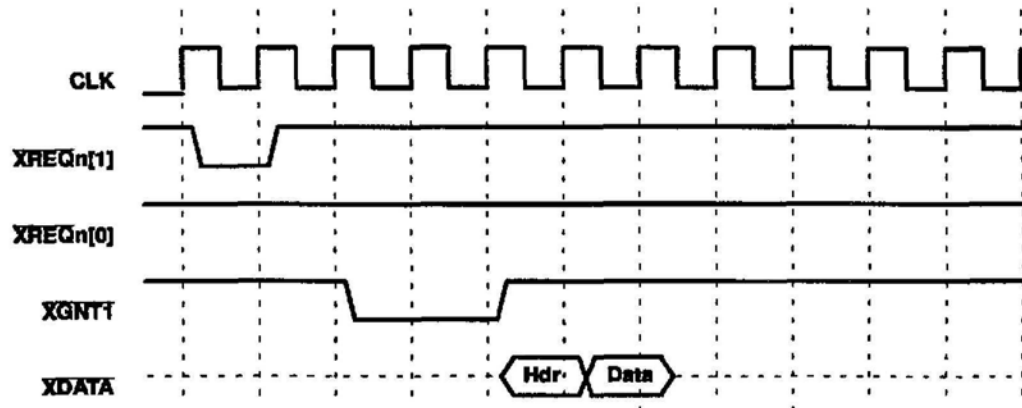


Figure 19–21 shows a nine-cycle packet transmitted by a bus watcher. The  $XREQn$ -lines are driven  $01_2$ , followed by another  $01_2$ . This is a request for a low-priority nine-cycle packet. The MXCC grants the bus to the BW for nine cycles.

Figure 19–21. XBus Nine-Cycle Packet

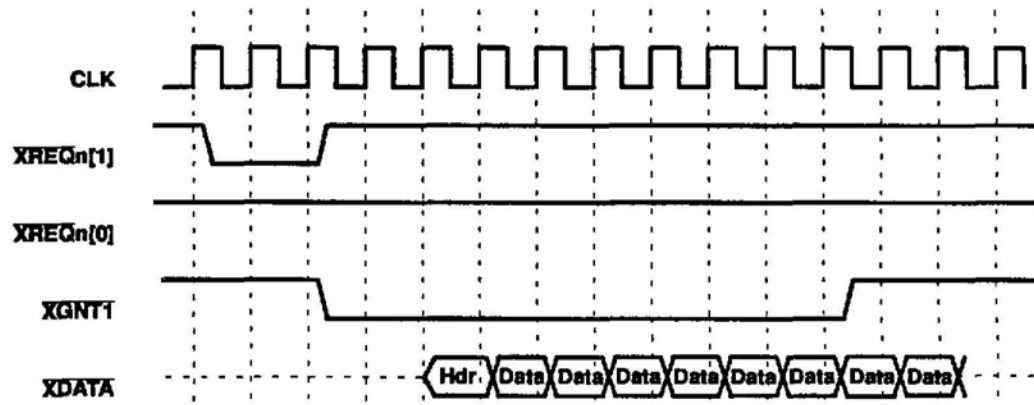


Figure 19–22 shows multiple BWs requesting the bus. BW0 requests the bus for a low-priority two-cycle packet, and BW1 requests the bus for a high-priority nine-cycle packet. The MXCC grants BW0 the bus for two cycles, then immediately grants BW1 the bus for nine cycles.

Figure 19–22. Multiple Requests

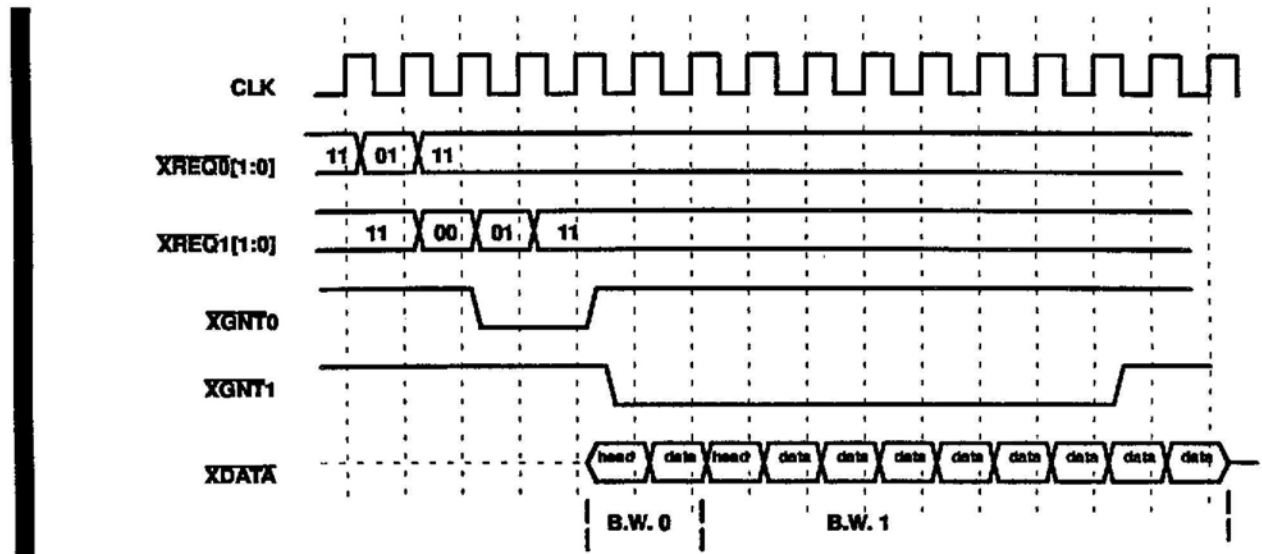


Figure 19–23 shows a TMS390Z55 two-cycle request packet, followed by a reply packet from the BW. Note that the XREQn arbitration request is for a low-priority two-cycle packet.

Figure 19–23. Two-Cycle Request and Two-Cycle Reply

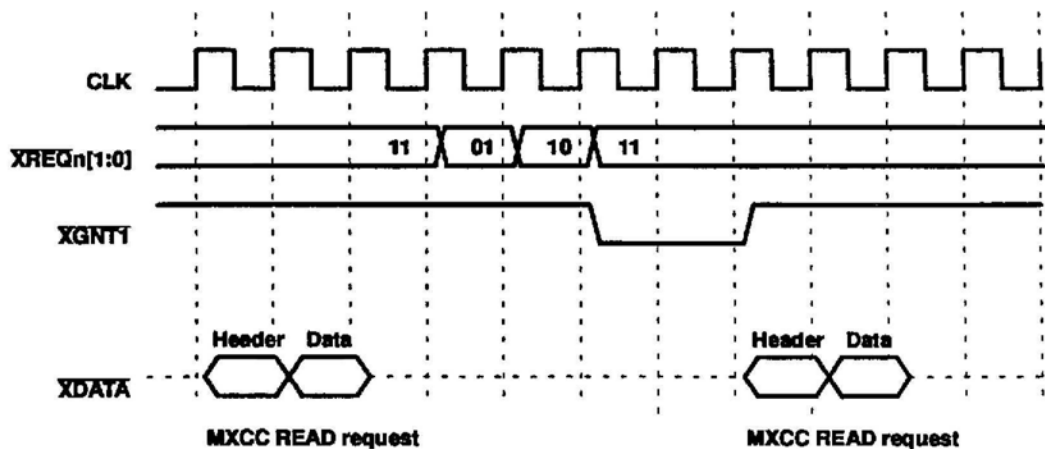


Figure 19–24 shows an MXCC two-cycle request packet, followed by a nine-cycle reply packet from the BW. The reply packet is high-priority.

Figure 19-24. Two-Cycle Request and Nine-Cycle Reply

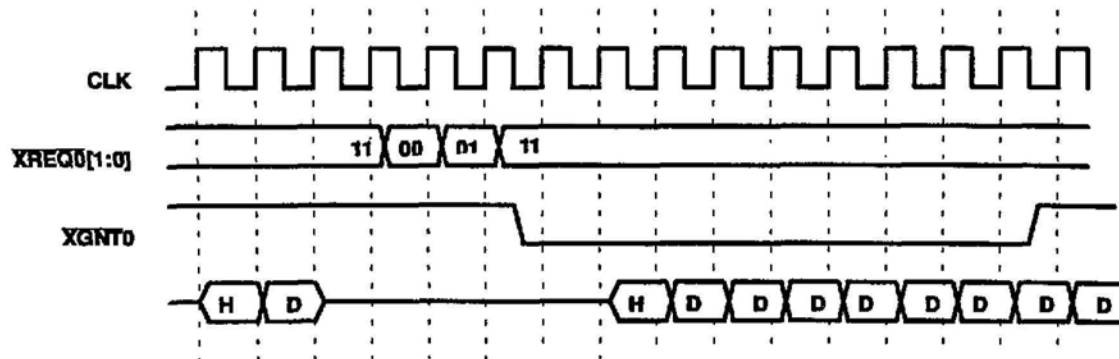
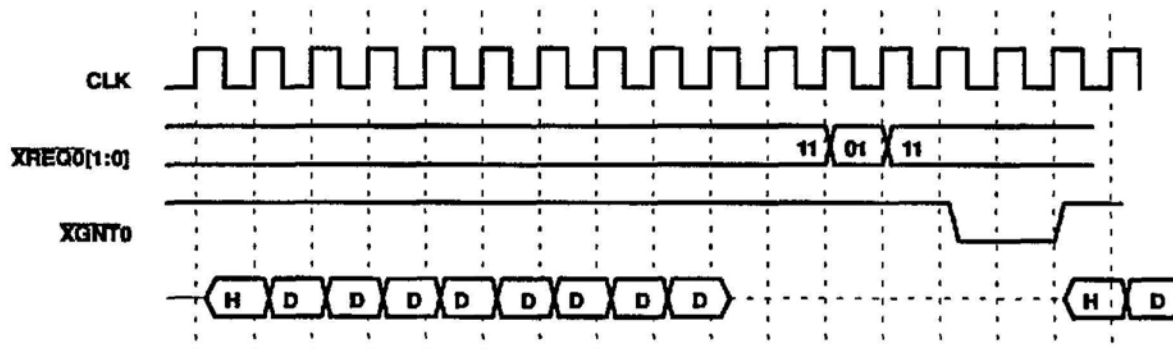


Figure 19-25 shows an MXCC nine-cycle packet (block write) and a corresponding two-cycle reply packet.

Figure 19-25. Nine-Cycle Request and Two-Cycle Reply





# **BootBus**

---

---

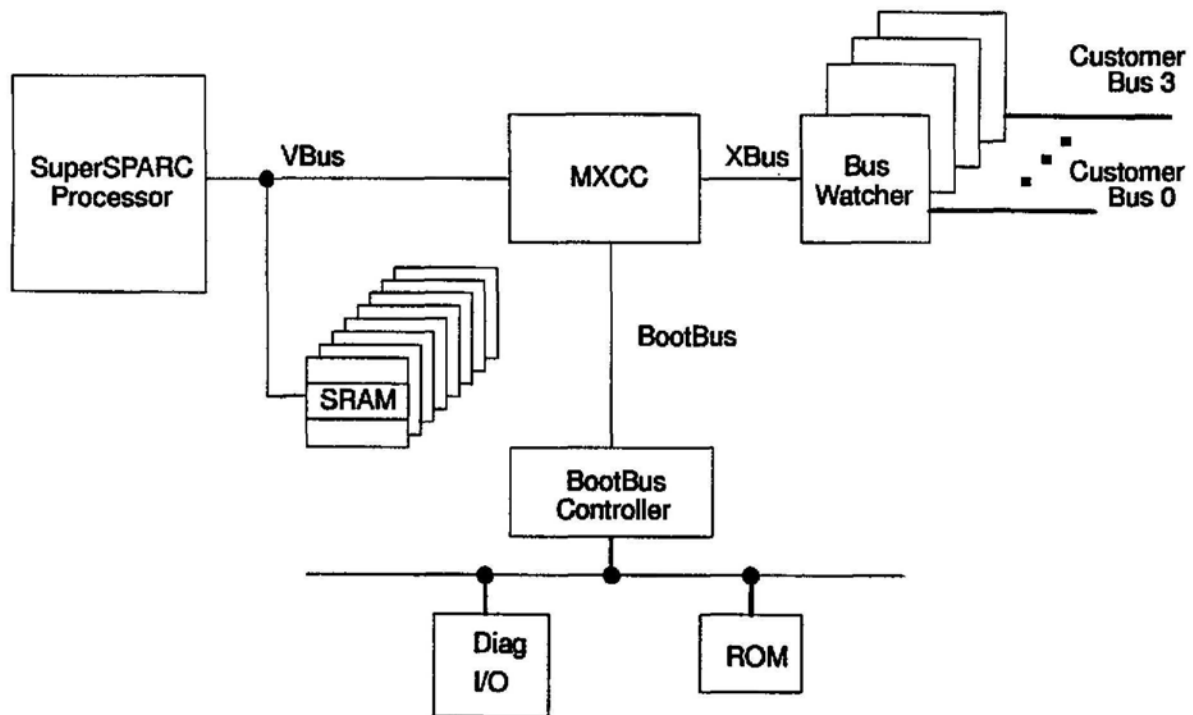
The BootBus is a simple synchronous 12-pin interface provided by the Multi-Cache Controller (MXCC) for accessing an EPROM for bootstrap loading and for accessing other low-speed peripherals. BootBus supports an address space of 16M-byte. Provisions are made for reading or writing from one to eight bytes from/to BootBus devices and for polling the devices for interrupts. BootBus is only available in the XBus configuration (when MBSEL is low). BootBus is accessible from both the VBus and the XBus.

<b>Topic</b>	<b>Page</b>
<b>20.1 Introduction</b> .....	<b>20-2</b>
<b>20.2 BootBus Signals</b> .....	<b>20-3</b>
<b>20.3 BootBus Addresses</b> .....	<b>20-5</b>
<b>20.4 BootBus Transactions</b> .....	<b>20-6</b>
<b>20.5 Multi-Byte Transfers</b> .....	<b>20-8</b>
<b>20.6 BootBus Example Transactions</b> .....	<b>20-9</b>

## 20.1 Introduction

A block diagram of the BootBus portion of a system is shown in Figure 20-1.

Figure 20-1. XBus System With BootBus

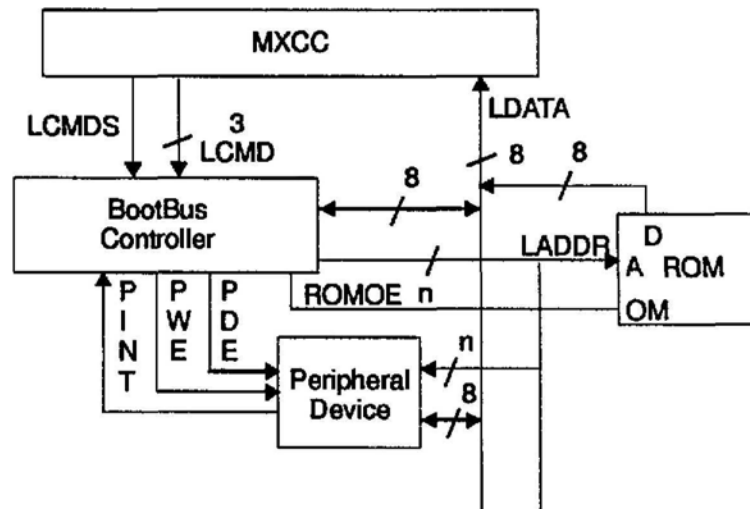


## 20.2 BootBus Signals

<b>LDATA[7:0]</b>	BootBus address, data, and interrupts according to the command on LCMD.
<b>LCMD[2:0]</b>	BootBus command. The commands are issued by the MXCC and interpreted by one or more of the external BootBus controllers. See Table 20-2 for the BootBus command encoding.
<b>LCMDS</b>	BootBus command strobe. When asserted, this signal indicates that command information on LCMD[2:0] and write data on LDATA[7:0], for WRITE-VALID commands, is valid. Input data is latched on the rising edge of LCMDS.

The connection of MXCC, the customer-designed BootBus controller, and example peripheral devices is shown in Example 20-1.

*Example 20-1. BootBus Connections*





The LCMD[2:0] signals connect from MXCC to the BootBus controller (BBC). The BBC decodes this command and addresses from LDATA[7:0] to control access to the devices on BootBus. The BBC latches the three portions of the address from ADR-HIGH, ADR-MED, and ADR-LOW commands and supplies the demultiplexed address to BootBus devices, such as memories, that require it. It may also decode a portion of the latched address to control which device is accessed for read and write operations. The BBC can use this information along with decoding LCMD[2:0] to control the output and write enables of the various BootBus devices.

Some BootBus devices may generate interrupts. The interrupt request signal from these devices are combined and encoded by the BBC. The result is used to reply to interrupt commands from the MXCC.

## 20.3 BootBus Addresses

Table 20-1 illustrates BootBus address decoding.

*Table 20-1. BootBus Address Decoding*

BUS	RANGE
VBus	Non-Cacheable Space ADDR[35:28] = 0xFF ADDR[27:24] = 0x0 or 0x1 ADDR[23:00] = BootBus Address
XBus	PA[35:28] = 0xFF PA[27:24] = 0x0 PA[23:00] = BootBus Address

PA = Physical Address X = don't care (MXCC will ignore)

BootBus is accessible only from the MXCC in XBus configurations. On the XBus, non-cacheable reads and writes with PA[24] set to zero access the BootBus. See Chapter 19, XBus.

The first instruction fetch by the SuperSPARC Processor (SSP) after reset is always at physical address 0xFF000000. In XBus configurations, this address always accesses BootBus. XBus systems must provide a read-only memory (ROM) or other source of instructions on BootBus for the system reset handler.

## 20.4 BootBus Transactions

### BootBus Commands

The encoding of LCMD[2:0] is shown in Table 20-2. All command cycles on BootBus are validated by LCMD5.

Table 20-2. BootBus Command Encoding

LCMD[2:0]	Name	Meaning
000	IDLE	idle
001	WRITE-VALID	data from MXCC
010	READ-VALID	data from device
011	IDLE-WR	idle for write
100	ADR-LOW	address[7:0]
101	ADR-MED	address[15:8]
110	INTERRUPT	interrupt status
111	ADR-HIGH	address[23:16]

☐ Idle Command

The idle command is used to three-state LDATA[7:0] whenever the driving source is changed. Both the MXCC and the BBC disable all drivers on LDATA during idle commands.

☐ Write Valid Command

The write valid command instructs the address decoder to write the selected device with the data on LDATA[7:0].

☐ Read Valid Command

The read valid command instructs the address decoder to drive the selected device data onto LDATA[7:0].

☐ Idle Write Command

The idle write command allows the BBC enough time to set up the targeted address. LDATA is three-stated during the idle write.

☐ Address Commands

Address low-byte, address middle-byte, and address high-byte commands are used to transmit portions of the BootBus address from the MXCC to the BBC. The MXCC uses three consecutive cycles on BootBus to send an address. The first cycle sends the high byte (bits 23:16) of the address. The second cycle sends the middle byte (bits 15:8) of the address. The third cycle sends the low byte (bits 7:0) of the address.

An external decoder must decode the address to select the appropriate device. This decoder is shown as the BBC in Figure 20-1.

☐ Interrupt Command

When the Interrupt command is asserted, the pending interrupt information encoded by the BBC is asserted on LDATA[3:0]. LDATA[7:4] are ignored by the MXCC during Interrupt commands. The encoding of interrupt information is as shown in Table 20-3.

Table 20-3. Interrupt Encoding

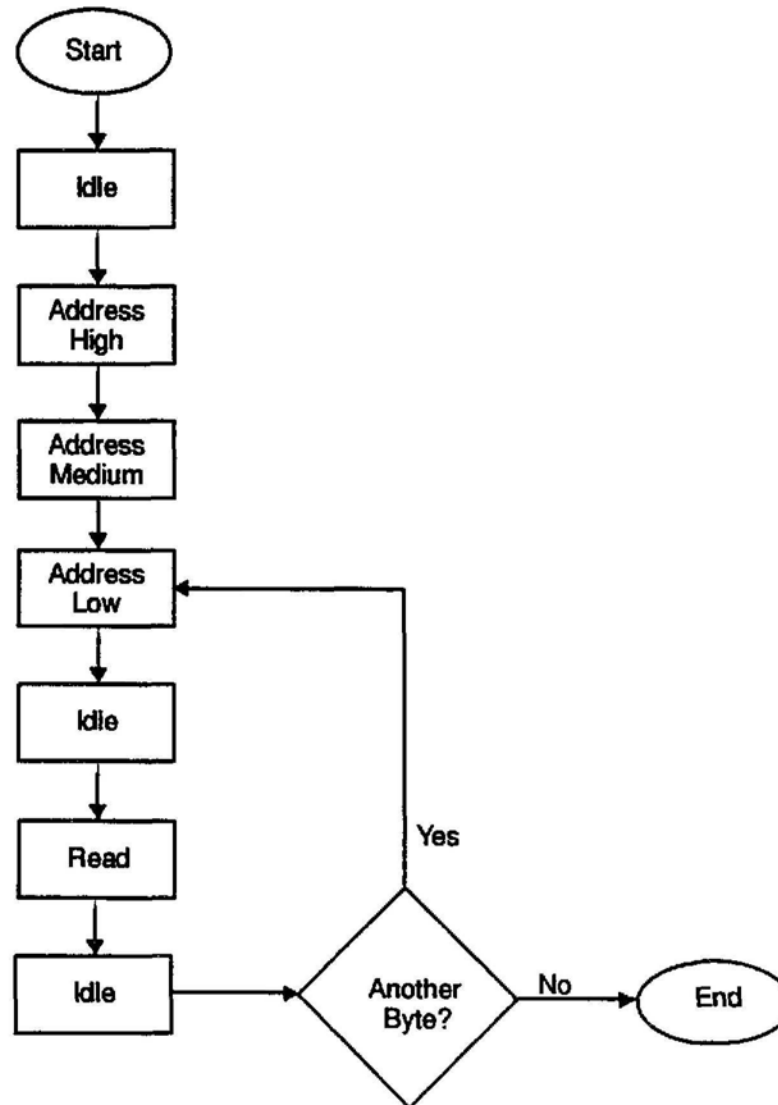
LDATA[3:0]	Encoding
0000	No pending interrupt
0001	Interrupt level 1
0010	Interrupt level 2
0011	Interrupt level 3
.	.
1110	Interrupt level 14
1111	Interrupt level 15

The MXCC sends Interrupt commands whenever BootBus either completes a Read or Write cycle or is not in use.

## 20.5 Multi-Byte Transfers

The MXCC can read or write multiple bytes on BootBus with an abbreviated addressing sequence between the constituent bytes. Figure 20-2 shows the steps of a multi-byte read on BootBus.

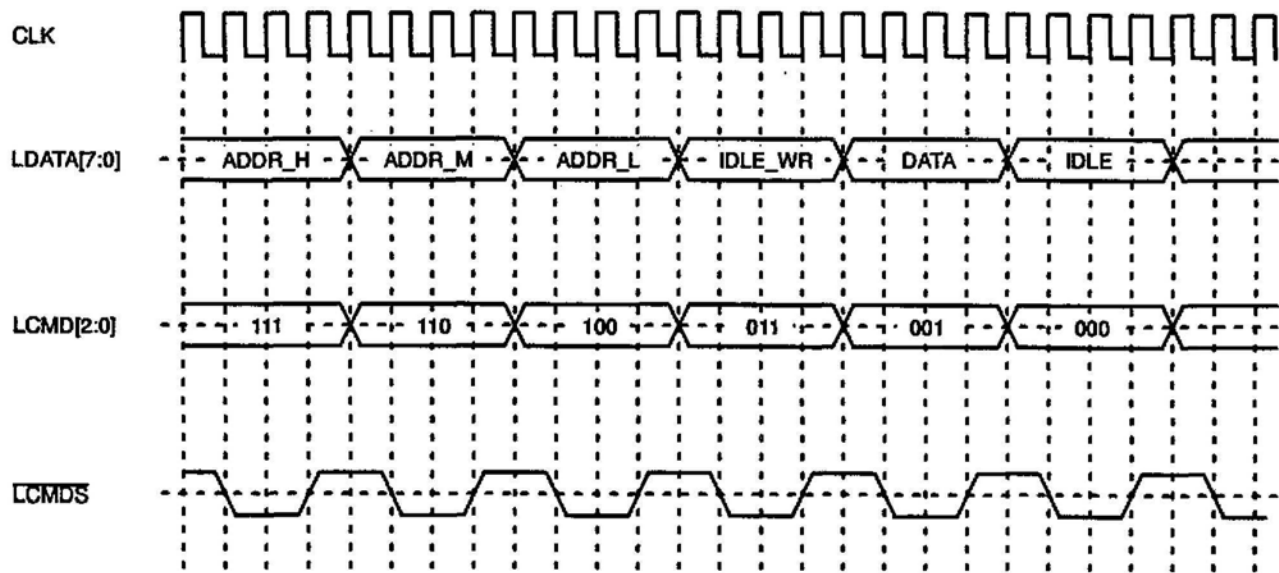
Figure 20-2. Steps in a Multi-Byte Read



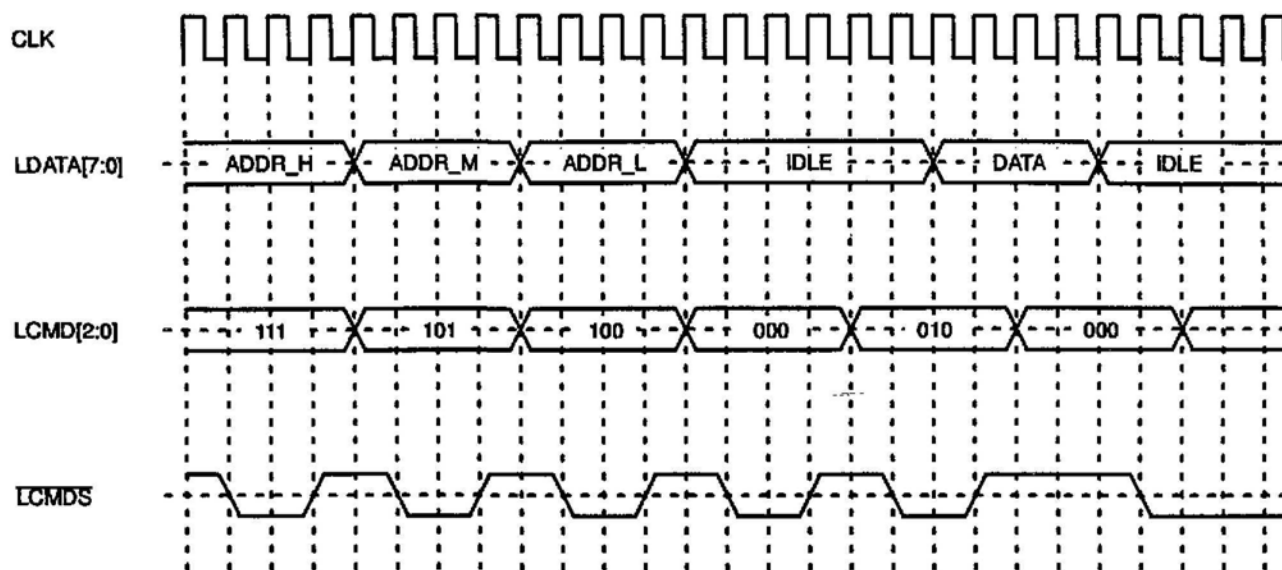
## 20.6 BootBus Example Transactions

Example 20-2, Example 20-3, Example 20-4, Example 20-5, and Example 20-6 are examples of BootBus transactions.

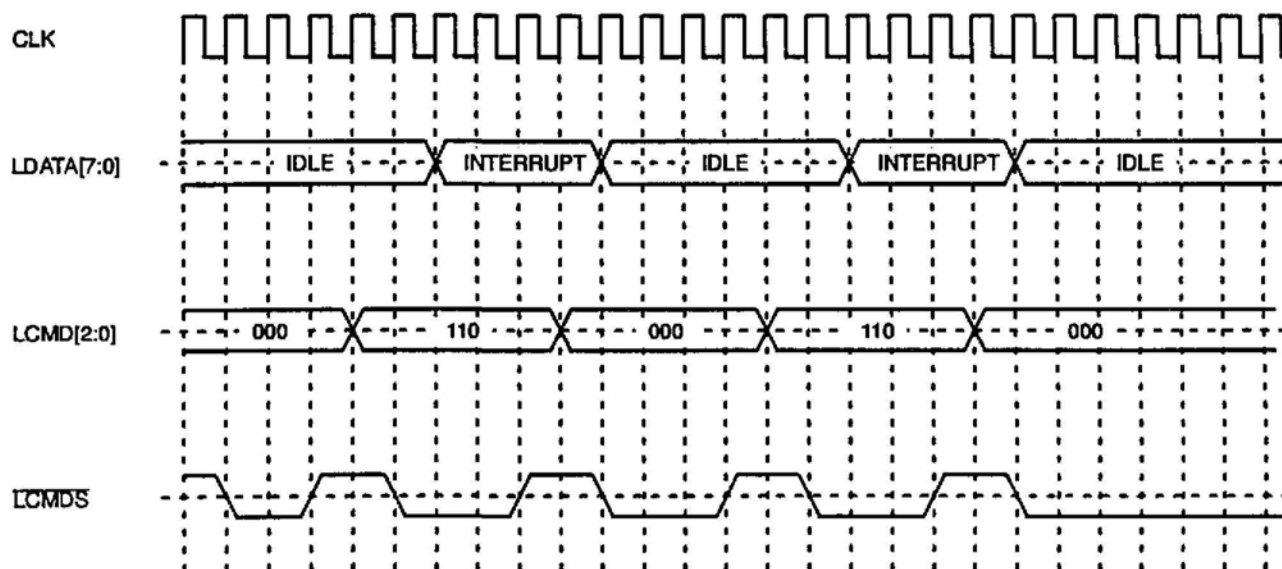
### Example 20-2. BootBus Write



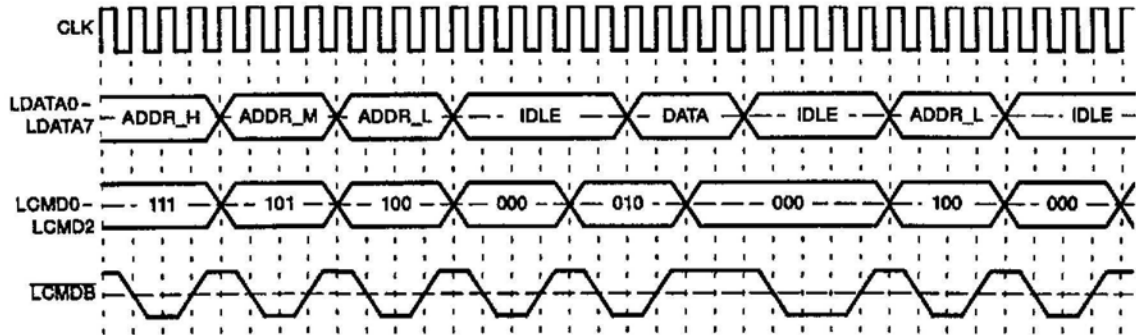
Example 20–3. BootBus Read



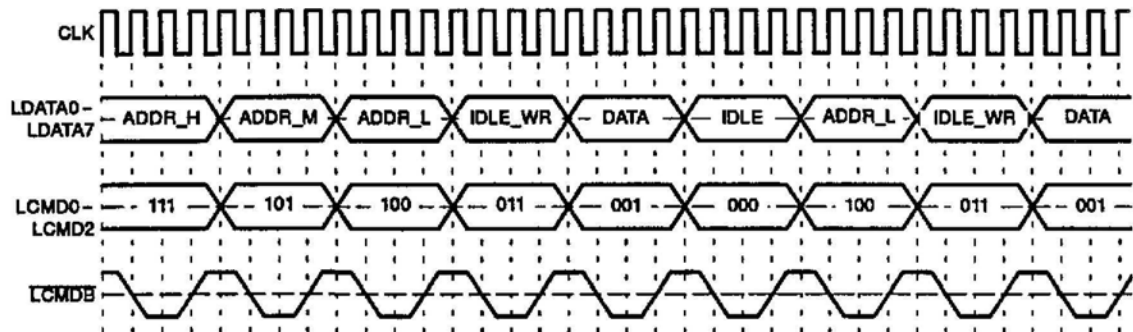
Example 20–4. BootBus Interrupt



**Example 20–5. Two-Byte BootBus Read**



**Example 20–6. Two-Byte BootBus Write**







# JTAG Serial Scan Interface

---

SuperSPARC provides the five-signal IEEE 1149.1 JTAG serial scan interface to allow observation and control for board-level, Built-In Self-Test (BIST), and chip-level testing; and to support a remote debugging environment.

Topic	Page
21.1 Introduction .....	21-2
21.2 JTAG Interface .....	21-3
21.3 TAP Controller .....	21-4
21.4 JTAG Operations .....	21-5
21.5 The SuperSPARC Processor JTAG Instructions .....	21-7
21.6 MultiCache Controller JTAG Instructions .....	21-14
21.7 System-Level Test .....	21-22

## 21.1 Introduction

The *IEEE 1149.1 Standard Access Port And Boundary Scan Architecture Specification* was developed to solve the problems of testing high-pin-count devices and to resolve system testing issues.

The SuperSPARC processor (SSP) implements the protocol defined in the *IEEE 1149.1 Standard Access Port and Boundary Scan Architecture Specification*. Users should be familiar with this specification before reading this chapter.

SuperSPARC provides the five-signal JTAG serial scan interface mechanism to support the following:

☐ **Manufacturing Fault Coverage**

JTAG allows test software access to internal scan logic to determine device manufacturing correctness.

☐ **Periphery and Interconnect Testing**

JTAG allows software-controlled boundary scan to test the periphery and interconnect between chips on boards that use SuperSPARC. Boundary scan testing requires software that uses JTAG to scan in, apply, scan out, and compare vectors.

☐ **Built-In Self-Test**

In addition to software initiation using ASI 0x39, SSP BIST can be initiated by software that uses the JTAG interface.

☐ **Remote Debugging Environment**

A scan-based debugger (SDB) can use JTAG to halt an application program, examine or alter register and memory state (including the program counters), set breakpoints or counters (to specify conditions where control should leave the application code and return to the SDB), and to resume control within the application code. Such software can download SPARC assembly language for the intended scan-based debug function, inspect device-specific JTAG status, and provide a recovery mechanism when scan-based debug instructions fault. See Chapter 22 for more details.

## 21.2 JTAG Interface

The IEEE 1149.1 JTAG serial-scan interface mechanism is composed of a set of pins and a test access port (TAP) controller state machine that responds to those pins. The design is partitioned into several serially scanned "rings" that are independently accessed. Ring selection is determined by the Instruction Register (IR). Access to the IR scan chain is in accordance with the JTAG protocol.

The SuperSPARC JTAG interface is composed of five pins:

- ☐ **Test Clock (TCK)**—The TCK signal is used to clock the TAP and the test data registers defined in this chapter.
- ☐ **Test Mode Select (TMS)**—The TMS signal is used by the TAP controller to move to other states. The TAP controller state diagram can be found in Section 5.1 of the *IEEE JTAG 1149.1 Specification*.
- ☐ **Test Logic Reset (TRST)**—TRST is used to reset the SuperSPARC internal JTAG TAP Controller.
- ☐ **Test Data In (TDI)**—Serially transmitted test instructions are sent via TDI.
- ☐ **Test Data Out (TDO)**—Data is serially transmitted from SuperSPARC to the external JTAG controller via TDO.

## 21.3 TAP Controller

The TAP controller is an internal sequencer that manages access to all JTAG test data registers. The TAP controller examines TRST and TMS sequences each TCK cycle for JTAG state transitions. The state of the TAP controls assertion of CAPTURE, SHIFT, and UPDATE operations. See the *IEEE 1149.1 Specification* for more details.

The TAP controller state diagram can be found in Section 5.1 of the *IEEE 1149.1 Specification*.

### 21.3.1 JTAG Reset Requirements

The TAP controller enters the TEST-LOGIC-RESET state when:

- ☐ TMS is asserted for five consecutive TCK cycles, or
- ☐ TRST is asserted for a single cycle.

Both TMS and TRST must be negated to exit the TEST-LOGIC-RESET state and move into the RUN-TEST/IDLE state.

The JTAG TAP controller must be reset at power-up to guarantee correct SuperSPARC operation. If TCK is not present, TRST must be asserted at power-up to properly reset the TAP controller. Otherwise, the JTAG IR will be in an indeterminate state that could result in undefined operations.

SuperSPARC requires that the external JTAG busmaster TAP controller remain in synchronization with SuperSPARC's internal JTAG TAP controller.

---

**Note:**

All systems (with TCK active) that use the TRST assertion to reset SuperSPARC's internal JTAG TAP controller must:

- ☐ Keep TMS asserted during TRST assertion.
  - ☐ Hold TMS asserted for a minimum of three TCK cycles (as seen by SuperSPARC) after negating TRST.
- 

---

**Note:**

If JTAG is not used in a system, TRST should be asserted to avoid unintended JTAG operation.

---

## 21.4 JTAG Operations

The three major scan chain operations defined in IEEE 1149.1 are:

- ☐ CAPTURE,
- ☐ SHIFT, and
- ☐ UPDATE.

Each of the serially scanned rings internal to SuperSPARC is composed of a reconfigurable shift register chain. Each stage of the scan chain has two register chains of equal length:

- ☐ The primary scan chain register.
- ☐ The update scan chain register.

The primary scan chain register is configurable as a shift register, while the update register is not. (See the *IEEE 1149.1 JTAG Specification* for more details). The functionality of a single-bit JTAG scan chain register is as shown in Figure 21-1.

Figure 21-1. JTAG Register

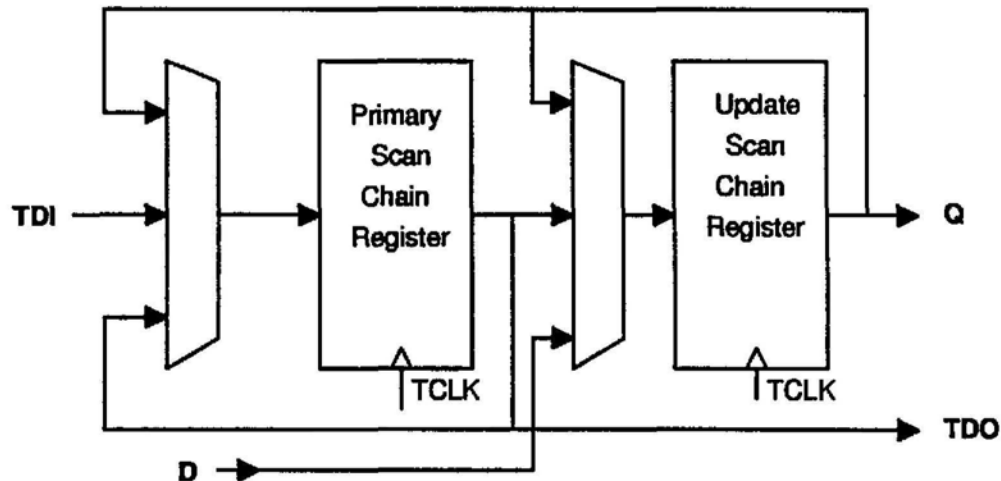
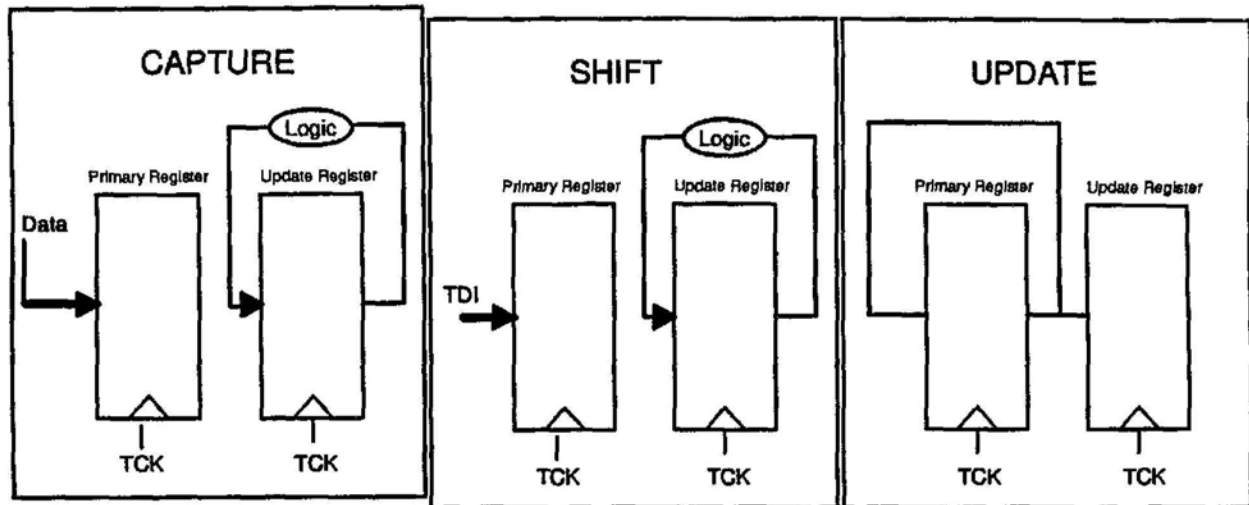


Figure 21-2 illustrates the three operations in a simplified functional view.

Figure 21-2. JTAG Operations—CAPTURE, SHIFT, and UPDATE



#### 21.4.1 JTAG CAPTURE Operation

During a CAPTURE operation, the JTAG scan chain element captures selected data into the primary register. After one TCK, those captured values are ready to be shifted out (using SHIFT operation) to the SuperSPARC level TDO at the end of the scan chain ring. In some specific cases, Capture-DR (Capture test data register) captures the value of a particular register into the primary register. This capability allows capturing of internal logic states, and the information is then shifted out to be read.

#### 21.4.2 JTAG SHIFT Operation

During a SHIFT operation, the JTAG scan chain element operates as a shift register. Each primary register stores its TDI value and shifts forward its TDO in the next TCK cycle as input to the next primary register in the scan chain. For a ring of size N, the TAP controller must shift N times to completely fill the scan chain. During this operation, the update register is unused and retains its value.

#### 21.4.3 JTAG UPDATE Operation

During an UPDATE operation, the JTAG scan chain element delivers the value contained in the primary register into the update register. Except for this overwrite period, the update register retains its previous value. Internal logic sees only the value contained in the update register. A single UPDATE cycle will deliver the intended values to the update register scan chain. The UPDATE is performed after all the values have been shifted into the primary scan chain registers. During this operation, the primary register retains its value. UPDATE is ignored by non-writable registers.

## **21.5 The SuperSPARC Processor JTAG Instructions**

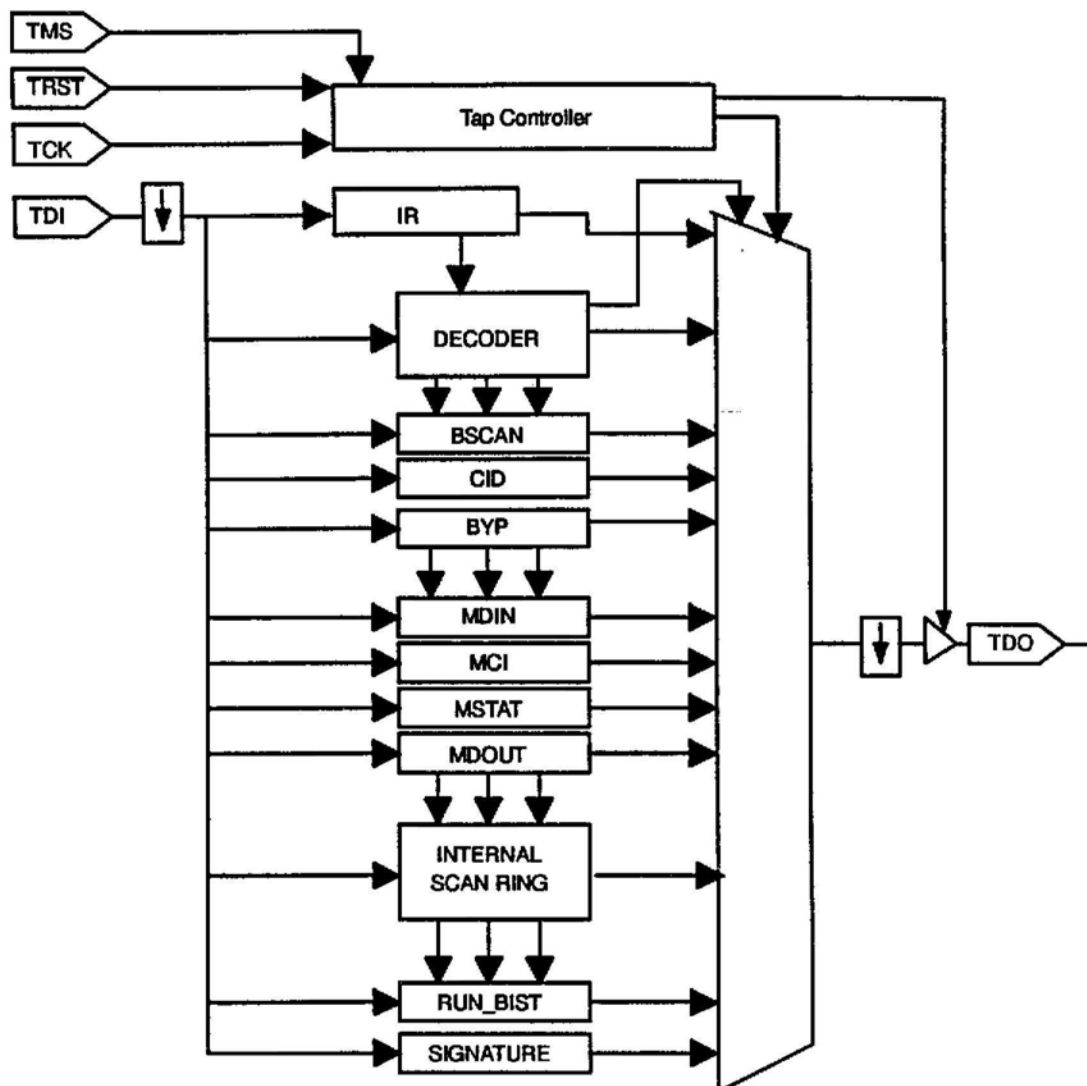
Five scan domains inside the SSP are accessible through JTAG:

- ☐ The IR register domain,
- ☐ The standard JTAG register domains (BYPASS, CID, BSCAN),
- ☐ An internal hardware test domain (internal use only),
- ☐ The BIST register domains (SHORT\_BIST, LONG\_BIST, SIGNATURE),  
and
- ☐ The scan-based debug register domains (MDIN, MCI, MSTAT, MDOUT).

Figure 21-3 is the block diagram of all JTAG-accessible serial-scan chains.



Figure 21-3. Block Diagram of JTAG Scan Chains Inside SuperSPARC



The SuperSPARC five-bit IR selects Test Data Register (TDR) scan chains for the SHIFT, UPDATE, and CAPTURE operations. The IR scan ring is selected when the JTAG TAP controller is in the UPDATE-IR state. A Capture-IR does not capture the current value of the IR update register. Instead, it returns a fixed binary encoding of 00001. The low two bits of this encoding are required by the IEEE 1149.1 Standard Access Port And Boundary Scan Architecture Specification.

The scan-based debug register domain consists of four scan chains: MDIN, MCI, MSTAT, and MDOUT. MDIN and MCI provide information from the SDB to the SSP, while MSTAT and MDOUT bring information from SuperSPARC to the SDB. See Chapter 22 for more details.

Internal scan rings are for manufacturing use only; other usage will provide undefined results.

The BIST register domain permits a BIST sequencer to internally generate, scan-in, apply, scan-out, and obtain a signature for state vectors.

The IR encoding to select access to a particular SSP TDR scan chain is shown in Table 21-1.

Table 21-1. TDR Scan Chain Selection by IR Encoding

TDR Ring Selected	IR Value	# bits
BSCAN_EXTTEST	0x00	290
BSCAN_SAMPLE	0x01	290
BSCAN_INTEST	0x02	290
Internal_Scan_Capture_Clock_Mode	0x03	n/a
Internal Scan Domain 0	0x04	n/a
Internal Scan Domain 1	0x05	n/a
Internal Scan Domain 2	0x06	n/a
Internal Scan Domain 3	0x07	n/a
MCI	0x08	37
MDIN	0x09	32
MDOUT	0x0a	32
MSTAT	0x0b	13
SIGNATURE	0x0c	32
RUN_SHORT_BIST	0x0d	n/a
RUN_LONG_BIST	0x0e	n/a
PLL	0x0f	n/a
CID	0x10-0x1e	32
BYPASS	0x1f	1

A brief description of each instruction is given below. For more details, see the *IEEE 1149.1 Standard Access Port And Boundary Scan Architecture Specification*.

## 21.5.1 BYPASS

The BYPASS scan chain comprises a one-bit primary scan register, with no update register. When IR selects BYPASS, the chip's TDI and TDO are essentially connected to this primary register. The Capture-DR state loads a one-bit zero into the primary register and Shift-DR requires one TCK cycle to forward data. The BYPASS register can be used to reduce the total scan chain length when other devices on the same TDI/TDO chain are being accessed. Update-DR has no effect on the BYPASS operation.

## 21.5.2 Component ID

The Component ID (CID) scan chain comprises a 32-bit ring of primary registers, where bit[0] is always 1, bit[11-1] the manufacturer ID, bit[27-12] the part number, and bit[31-28] the version number. (See Figure 21-4.) Capture-DR loads the SuperSPARC component ID into the CID primary register's scan chain. Subsequent Shift-DR cycles shift (bit[0]) out and scan new TDI data in (bit[31]). There is no update register, so Update-DR has no effect.

Figure 21-4. JTAG ID Register Format



<b>Ver</b>	Version number. Incremented on component revisions.
<b>pnum</b>	Part Number. A component ID assigned by the manufacturer. The SSP has a value of 0x04 in this field.
<b>manId</b>	Manufacturer ID. The identification number of the component's manufacturer. This field is set to 0x17 for Texas Instruments.

## 21.5.3 Boundary Scan (BSCAN)

SuperSPARC BSCAN consists of three operations: BSCAN-EXTEST, BSCAN-SAMPLE, and BSCAN-INTEST. The SSP BSCAN is a 290-bit ring of JTAG scan chain register elements. Capture-DR reads data from the chip pins into the primary register. Shift-DR forwards data through the scan chain, where bit[289] is output to TDO. (See note below.) For the entire chain to be completely written in or read out, 290 TCK cycles are needed. Update-DR copies data from the primary register into the update register, requiring only one TCK cycle. Table 21-2 portrays the SSP's boundary scan map.

### Note:

SSP revisions 1.X, 2.X, and 3.X have a BSCAN chain length of 290 bits. Future revisions may have different BSCAN chain lengths.

The SAMPLE/PRELOAD instruction is used to preload data into the BSCAN chain for use by other instructions and to sample the data flowing through the SSP pins. The SAMPLE portion of the instruction occurs on the rising edge of TCK during the Capture-DR state, while the PRELOAD occurs during the Update-DR state on the rising edge of TCK. The PRELOAD data is scanned in while the SAMPLE data is scanned out.

When the EXTEST instruction is selected, data that has been loaded into the BSCAN chain and, corresponding to output cells, will be forced onto the SSP output pins on the falling edge of TCK in the Update-IR state. This data will change only on the falling edge of TCK in the Update-DR state. This instruction allows testing of board-level interconnections. When this instruction is selected, all signals received at the SSP input pins will be loaded into the BSCAN chain in the Capture-DR state.

Selection of the INTEST instruction allows testing of the SSP's internal logic. Data that has been loaded into the BSCAN chain and corresponding to input pins will be forced into the SSP's logic during the Update-IR state. This data will change only on the falling edge of TCK in the Update-DR state. Outputs of the logic will be captured into the BSCAN register in the Capture-DR state. This data can then be scanned out for analysis.

#### 21.5.4 SHORT\_BIST, LONG\_BIST

The SuperSPARC BIST mechanism is initiated by or examined through either JTAG or ASI memory references. When BIST is initiated, any pre-BIST SuperSPARC state will be destroyed. At the completion of a JTAG-initiated BIST, the user needs to generate the reset. This can be done by entering the TAP reset state by either assertion of TMS for five consecutive TCK cycles or asserting TRST. Internal timing sequencing will guarantee PLL restabilization. Once initiated, BIST will be under the control of SuperSPARC, and the JTAG TAP controller need not remain in WAIT/RUN\_BIST state. See Section 13.4 for more details. UPDATE has no effect. At the completion of BIST, a CAPTURE of the SIGNATURE TDR should be done.

**Note:**

LONG\_BIST operation is not tested during manufacturing test.

#### 21.5.5 Signature

The signature scan chain is a 31-bit ring. Both long and short BIST operations cause the BIST sequencer to generate, scan in, apply, and scan out a signature for one of two pre-defined pseudo-random test vectors. The SuperSPARC response to these vectors is collected and compressed into the signature register. The correct value of the signature register will be different for these two cases. See Section 13.4 for more details.

Table 21-2. SuperSPARC Boundary Scan Bit Definition

PIN	SIGNAL	PIN	SIGNAL	PIN	SIGNAL	PIN	SIGNAL	PIN	SIGNAL	PIN	SIGNAL	PIN	SIGNAL	PIN	SIGNAL
0	reset_in	40	data45_in	80	dpwr0_in	120	addr10_out	160	wee_out	200	data6_in	240	data26_in	280	addr31_in
1	test_in	41	data45_out	81	dpwr0_out	121	addr09_out	161	mexc_in	201	data6_out	241	data26_out	281	addr31_out
2	srmta_in	42	data44_in	82	pend_in	122	addr09_in	162	ardy_in	202	data7_in	242	data27_in	282	addr32_in
3	ccrdy_in	43	data44_out	83	size1_out	123	addr09_out	163	ardy_out	203	data7_out	243	data27_out	283	addr32_out
4	data83_in	44	vck_in	84	size0_out	124	addr08_in	164	busreq_out	204	data8_in	244	data28_in	284	addr33_in
5	data63_out	45	plibyp_in	85	owner_in	125	addr08_out	165	wrdy_in	205	data8_out	245	data28_out	285	addr33_out
6	data62_in	46	data43_in	86	owner_out	126	dpwr0_out	166	wrdy_out	206	data9_in	246	data29_in	286	addr34_in
7	data62_out	47	data43_out	87	shared_in	127	addr07_in	167	rrdy_in	207	data9_out	247	data29_out	287	addr34_out
8	data61_in	48	data42_in	88	shared_out	128	addr07_out	168	wgrt_in	208	data10_in	248	data30_in	288	addr35_in
9	data61_out	49	data42_out	89	error_out	129	srboe_out	169	rgrt_in	209	data10_out	249	data30_out	289	addr35_out
10	data60_in	50	data41_in	90	cchbl_out	130	addr06_in	170	cmda_in	210	data11_in	250	data31_in		
11	data60_out	51	data41_out	91	su_out	131	addr06_out	171	cmda_out	211	data11_out	251	data31_out		
12	data59_in	52	data40_in	92	addr23_in	132	alibo_out	172	demap_in	212	data12_in	252	pipe00_out		
13	data59_out	53	data40_out	93	addr23_out	133	erboe_out	173	demap_out	213	data12_out	253	pipe01_out		
14	data58_in	54	data39_in	94	addr22_in	134	addr05_in	174	csa_out	214	data13_in	254	pipe02_out		
15	data58_out	55	data39_out	95	addr22_out	135	addr05_out	175	ldst_out	215	data13_out	255	pipe03_out		
16	data57_in	56	data38_in	96	addr21_in	136	addr04_in	176	dpwr4_in	216	data14_in	256	pipe04_out		
17	data57_out	57	data38_out	97	addr21_out	137	addr04_out	177	dpwr4_out	217	data14_out	257	pipe05_out		
18	data56_in	58	data37_in	98	addr20_in	138	srboe_out	178	dpwr5_in	218	data15_in	258	pipe06_out		
19	data56_out	59	data37_out	99	addr20_out	139	addr03_in	179	dpwr5_out	219	data15_out	259	pipe07_out		
20	data55_in	60	data36_in	100	addr19_in	140	dpwr0_out	180	dpwr6_in	220	data16_in	260	pipe08_out		
21	data55_out	61	data36_out	101	addr19_out	141	mbboe_out	181	dpwr6_out	221	data16_out	261	pipe09_out		
22	data54_in	62	data35_in	102	addr18_in	142	addr02_in	182	dpwr7_in	222	data17_in	262	ir0_in		
23	data54_out	63	data35_out	103	addr18_out	143	addr02_out	183	dpwr7_out	223	data17_out	263	ir1_in		
24	data53_in	64	data34_in	104	addr17_in	144	mirboe_out	184	we4_out	224	data18_in	264	ir2_in		
25	data53_out	65	data34_out	105	addr17_out	145	srboe_out	185	we5_out	225	data18_out	265	ir3_in		
26	data52_in	66	data33_in	106	addr16_in	146	addr01_in	186	we6_out	226	data19_in	266	addr24_in		
27	data52_out	67	data33_out	107	addr16_out	147	addr01_out	187	we7_out	227	data19_out	267	addr24_out		
28	data51_in	68	data32_in	108	addr15_in	148	addr00_in	188	data0_in	228	data20_in	268	addr25_in		
29	data51_out	69	data32_out	109	addr15_out	149	addr00_out	189	data0_out	229	data20_out	269	addr25_out		
30	data50_in	70	we3_out	110	addr14_in	150	oe_in	190	data1_in	230	data21_in	270	addr26_in		
31	data50_out	71	we2_out	111	addr14_out	151	oe_out	191	data1_out	231	data21_out	271	addr26_out		
32	data49_in	72	we1_out	112	addr13_in	152	wr_in	192	data2_in	232	data22_in	272	addr27_in		
33	data49_out	73	we0_out	113	addr13_out	153	wr_out	193	data2_out	233	data22_out	273	addr27_out		
34	data48_in	74	dpwr3_in	114	addr12_in	154	oeboe_out	194	data3_in	234	data23_in	274	addr28_in		
35	data48_out	75	dpwr3_out	115	addr12_out	155	rd_in	195	data3_out	235	data23_out	275	addr28_out		
36	data47_in	76	dpwr2_in	116	deboe_out	156	rd_out	196	data4_in	236	data24_in	276	addr29_in		
37	data47_out	77	dpwr2_out	117	addr11_in	157	buret_out	197	data4_out	237	data24_out	277	addr29_out		
38	data46_in	78	dpwr1_in	118	addr11_out	158	retry_in	198	data5_in	238	data25_in	278	addr30_in		
39	data46_out	79	dpwr1_out	119	addr10_in	159	wee_in	199	data5_out	239	data25_out	279	addr30_out		

**Note:**

This table reflects SSP revisions 1.X, 2.X, and 3.X. Future SSP revisions may have different BSCAN chain lengths and configurations.

### 21.5.6 Scan-Based Debug

There are four independent scan rings associated with scan-based debug:

- ☐ MDIN,
- ☐ MCI,

☐ MSTAT, and

☐ MDOUT.

The MDIN scan chain is a 32-bit ring. Shift-DR shifts data from a primary register into the next primary register. Bit[0] goes out to TDO, while bit[31] reads in TDI. Update-DR copies data from the primary register into the update register. Capture-DR has no effect.

The MCI scan chain is a 37-bit ring. Shift-DR shifts the primary register to the next primary register in the chain, where bit[36] reads in TDI and bit[0] outputs TDO. Update-DR copies data from the primary register into the update register.

The MSTAT scan chain is a 13-bit ring. Capture-DR captures data from the MSTAT register into the primary register. Shift-DR shifts the primary register to the next primary register in the chain, where bit[12] reads TDI and bit[0] outputs TDO. Update-DR has no effect.

The MDOUT scan chain is a 32-bit ring. Capture-DR captures data from the MDOUT register into the primary register. Shift-DR shifts the primary register to the next primary register in the chain, where bit[31] reads TDI and bit[0] outputs TDO. Update-DR has no effect. See Chapter 22 for more details on scan-based debug.

### **21.5.7 SEE\_PLL**

This scan chain is used to verify the integrity of the on-chip PLL with respect to clock jitter and VCO behavior. When selected, the scan chain will output PLL clock on TDO. This scan chain is used for hardware manufacturing tests and should not be used when the SSP is plugged into a board.

### **21.5.8 INTERNAL\_SCAN**

This scan chain is used for hardware manufacturing tests; details are not provided in this manual.

## 21.6 MultiCache Controller JTAG Instructions

Four scan domains inside the MultiCache Controller (MXCC) are accessible via JTAG:

- ☐ The IR domain,
- ☐ The standard JTAG register domain (BYPASS, CID, BSCAN),
- ☐ An internal hardware test domain (for internal use only), and
- ☐ The status scan domain (DATASCAN, BCSCAN).

The MXCC-IR four-bit serial register selects the test data register scan chains for the SHIFT, UPDATE, and CAPTURE operations. The IR scan ring is selected when the JTAG TAP controller is in the Update IR state. A capture IR returns the fixed binary value of 0001 as specified by the *IEEE 1149.1 Standard Access Port And Boundary Scan Architecture Specification*.

The IR encoding for each scan chain supported by MXCC is shown in Table 21-3.

Table 21-3. JTAG Instruction register encoding

IR Value	Instruction	Summary
0x0	EXTEST	Boundary Scan
0x1	SAMPLE/PRELOAD	Boundary Scan
0x2	INTEST	Boundary Scan
0x3	INTSHIFT	Internal F/F observability
0x4	INTSCAN	Internal Scan
0x5	DATASCAN	Status Scan
0x6	CID	Device Ident
0x7	BCSCAN	Boot Comm
0x8 -- 0xE	reserved	N/A
0xF	BYPASS	Bypass Scan

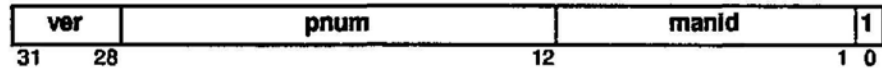
### 21.6.1 BYPASS

The BYPASS instruction selects the Bypass Test Data Register, which is a single-bit scan register. The Bypass register can be used to reduce total scan chain length when other devices on the same TDI/TDO chain are being accessed. When there are many chips on a board and in the same chain and only a few chips need to be accessed, the other chips should have the Bypass register selected. The BYPASS instruction can be scanned into all the other devices, thereby considerably reducing the length of the overall scan chain. When the Bypass test data register is selected, a single bit of logic zero will be sent to TDO in the Shift-DR state. Thereafter, TDO will follow TDI delayed by one TCK.

### 21.6.2 COMPONENT ID (CID)

The CID instruction selects the CID Test Data Register, which is a 32-bit scan register. The CID register has the format shown in Figure 21-5 and contains the MXCC's assigned JTAG component identifier. This register can only be captured—never updated.

Figure 21-5. JTAG ID register format



<b>ver</b>	Version number. Incremented on component revisions.
<b>pnum</b>	Part Number. A component ID assigned by the manufacturer. The MXCC has a value of 0x03 in this field.
<b>manid</b>	Manufacturer ID. The identification number of the component's manufacturer. This field is set to 0x17 for Texas Instruments.

### 21.6.3 Boundary Scan (BSCAN)

The BSCAN TDR of the MXCC is 485 bits long (see note below). This register supports three JTAG instructions:

- ☐ SAMPLE/PRELOAD,
- ☐ EXTEST, and
- ☐ INTEST.

Table 21-5 is the map of the MXCC boundary scan chain in the MBus configuration. Table 21-6 is the map in the XBus configuration. Bit[484] is connected



to TDO; bit[0] is connected to TDI. Update-DR copies data from the primary register into the update register.

The SAMPLE/PRELOAD instruction is used to preload data into the BSCAN chain for use by other BSCAN instructions and to sample the data flowing through the MXCC pins. The SAMPLE portion of the instruction occurs on the rising edge of TCK during the Capture-DR state, while the PRELOAD occurs during the Update-DR state on the rising edge of TCK. The PRELOAD data is scanned in while the SAMPLE data is scanned out.

When the EXTEST instruction is selected, data that has been loaded into the BSCAN chain and corresponding to output cells will be forced onto the MXCC output pins on the falling edge of TCK in the Update IR state. This data will change only on the falling edge of TCK in the Update DR state. This instruction allows testing of board-level interconnections. When this instruction is selected, all signals received at the MXCC input pins will be loaded into the boundary scan chain in the Capture-DR state.

Selection of the INTEST instruction allows testing of the MXCC's internal logic. Data that has been loaded into the BSCAN chain and corresponding to input pins will be forced into the MXCC's logic during the Update IR state. This data will change only on the falling edge of TCK in the Update DR state. Outputs of the logic will be captured into the BSCAN register in the Capture-DR state. This data can then be scanned out for analysis.

---

**Note:**

Rev. 2.X MXCC BSCAN chain is 485 bits. Rev. 1.X MXCC BSCAN chain is 483 bits. Future revisions may have different BSCAN chain lengths.

---

#### 21.6.4 DATASCAN

The DATASCAN instruction selects the Datascan TDR. This register consists of the MXCC Control Register, Reset Register, Error Register, Status Register, and BIST signature register. When this register is selected, all of these registers may be captured and scanned out during normal operation of the MXCC. This register may be captured but not updated. The format of the Datascan register is shown in Table 21-4.

#### 21.6.5 BCSCAN

The BCSCAN TDR consists of a one-bit primary register. A JTAG master can use this instruction to update the contents of the MXCC STATUS. BC bit while the system is running to communicate with the SSP.

#### 21.6.6 INTSCAN/INTSHIFT

This scan chain is used for hardware manufacturing tests; details are not provided in this manual.

Table 21-4. DATASCAN Register Format

Bit #	Parameter	Register
0	SI (Software Internal Reset)	Reset
1	WD (Watchdog Reset)	Reset
2	PP (Prefetch Pending)	Status
3	RP (Read Pending)	Status
4	WP (Write Miss Pending)	Status
5	BC (Boot Communication)	Status
6-9	SPC[0:3] (Store Pending Count)	Status
10-13	NCSPC[0:3] (NC Store Pending Count)	Status
14-37	NCSPA[0:23] (NC Store Page Address)	Status
38-39	NCSID[0:1] (NC Store BW ID)	Status
40	SM (Synchronous Mode)	Status
41	SXP (Store Exception Pending)	Status
42	HC (Half Cache)	Control
43	CS (E-cache Size)	Control
44	CE (E-cache Enable)	Control
45	PE (Parity Enable)	Control
46	MC (Multiple Cmd Enable)	Control
47	PF (Prefetch Enable)	Control
48	WI (Write Invalidation)	Control
49-50	BWC[0:1] (Bus Watcher Count)	Control
51	RC (Read Reference Count)	Control
52-82	BIST Signature [30:0]	Bist
83-118	PA[0:35] (Physical Address)	Error
119-120	0b00 (reserved, reads zero)	Error
121	S (Supervisor Mode)	Error
122-129	ERR[0:7] (Error Code)	Error
130-139	CCOP[0:9] (Cache Controller Operation Code)	Error
140	EV (Error Information Valid)	Error
141	AE (Asynchronous Error)	Error
142	CP (Parity Error, MXCC Master)	Error
143	VP (Parity Error, SuperSPARC Master)	Error
144	CC (Cache Consistency Error)	Error
145	XP (XBus Parity Error)	Error
146	ME (Multiple Errors)	Error

Table 21–5. MXCC JTAG Boundary Scan Bit Order for MBus

1	I	MBSEL	43	O	MAD06	85	I	MAD24	127	I	—	169	O	MAD47	211	O	IRL1
2	I	MIRL1	44	I	MAD07	86	O	MAD24	128	I	—	170	E	oe-mxd10	212	O	IRL2
3	O	MIRL1	45	O	MAD07	87	I	MAD25	129	I	—	171	E	oe-mxd11	213	O	IRL3
4	I	MIRL0	46	E	oe-mxd0	88	O	MAD25	130	O	MBF	172	I	MAD48	214	I	ADDR24
5	O	MIRL0	47	E	oe-mxd1	89	I	MAD26	131	I	MID3	173	O	MAD48	215	O	ADDR24
6	I	MIRL3	48	I	MAD08	90	O	MAD26	132	O	MID3	174	I	MAD49	216	I	ADDR25
7	O	MIRL3	49	O	MAD08	91	I	MAD27	133	I	MID0	175	O	MAD49	217	O	ADDR25
8	I	MIRL2	50	I	MAD09	92	O	MAD27	134	O	MID0	176	I	MAD50	218	I	ADDR26
9	O	MIRL2	51	O	MAD09	93	I	MAD28	135	O	—	177	O	MAD50	219	O	ADDR26
10	I	—	52	I	GTLREF1	94	O	MAD28	136	I	MAD32	178	I	MAD51	220	I	ADDR27
11	O	—	53	I	MAD10	95	I	MAD29	137	O	MAD32	179	O	MAD51	221	O	ADDR27
12	I	—	54	O	MAD10	96	O	MAD29	138	I	MAD33	180	I	MAD52	222	I	ADDR28
13	O	—	55	I	MAD11	97	I	MAD30	139	O	MAD33	181	O	MAD52	223	O	ADDR28
14	I	—	56	O	MAD11	98	O	MAD30	140	I	MAD34	182	I	MAD53	224	I	ADDR29
15	O	—	57	I	MAD12	99	I	MAD31	141	O	MAD34	183	O	MAD53	225	O	ADDR29
16	I	—	58	O	MAD12	100	O	MAD31	142	I	MAD35	184	I	MAD54	226	I	ADDR30
17	O	—	59	I	MAD13	101	E	oe-mxd6	143	O	MAD35	185	O	MAD54	227	O	ADDR30
18	E	oe-tb-dt	60	O	MAD13	102	E	oe-mxd7	144	I	MAD36	186	I	MAD55	228	E	oe-addr3
19	O	—	61	I	MAD14	103	I	MERR	145	O	MAD36	187	O	MAD55	229	I	ADDR31
20	O	—	62	O	MAD14	104	O	MERR	146	I	MAD37	188	E	oe-mxd12	230	O	ADDR31
21	I	MID1	63	I	MAD15	105	E	oe-merr	147	O	MAD37	189	E	oe-mxd13	231	I	ADDR32
22	O	MID1	64	O	MAD15	106	I	MIDY	148	I	MAD38	190	I	MAD56	232	O	ADDR32
23	I	MID2	65	E	oe-mxd2	107	O	MIDY	149	O	MAD38	191	O	MAD56	233	I	ADDR33
24	O	MID2	66	E	oe-mxd3	108	E	oe-midy	150	I	MAD39	192	I	MAD57	234	O	ADDR33
25	O	AERR	67	I	MAD16	109	I	MRTY	151	O	MAD39	193	O	MAD57	235	I	ADDR34
26	E	oe-aerr	68	O	MAD16	110	O	MRTY	152	E	oe-mxd8	194	I	MAD58	236	O	ADDR34
27	I	spare-in	69	I	MAD17	111	I	MBC	153	E	oe-mxd9	195	O	MAD58	237	I	ADDR35
28	O	spare-out	70	O	MAD17	112	O	MBC	154	I	MAD40	196	I	MAD59	238	O	ADDR35
29	I	RSTIN	71	I	MAD18	113	E	oe-xpar	155	O	MAD40	197	O	MAD59	239	I	PCLK
30	I	MAD00	72	O	MAD18	114	I	MBB	156	I	MAD41	198	I	MAD60	240	I	DATA63
31	O	MAD00	73	I	MAD19	115	O	MBB	157	O	MAD41	199	O	MAD60	241	O	DATA63
32	I	MAD01	74	O	MAD18	116	E	oe-mbb	158	I	MAD42	200	I	MAD61	242	I	DATA62
33	O	MAD01	75	I	MAD20	117	I	MAS	159	O	MAD42	201	O	MAD61	243	O	DATA62
34	I	MAD02	76	O	MAD20	118	O	MAS	160	I	MAD43	202	I	MAD62	244	I	DATA61
35	O	MAD02	77	I	MAD21	119	E	oe-mas	161	O	MAD43	203	O	MAD62	245	O	DATA61
36	I	MAD03	78	O	MAD21	120	I	MIF	162	I	MAD44	204	I	MAD63	246	I	DATA60
37	O	MAD03	79	I	MAD22	121	O	MIF	163	O	MAD44	205	O	MAD63	247	O	DATA60
38	I	MAD04	80	O	MAD22	122	E	oe-mih	164	I	MAD45	206	E	oe-mxd14	248	I	DATA59
39	O	MAD04	81	I	MAD23	123	I	MSH	165	O	MAD45	207	E	oe-mxd15	249	O	DATA59
40	I	MAD05	82	O	MAD23	124	O	MSH	166	I	MAD46	208	I	BCLK	250	I	DATA58
41	O	MAD05	83	E	oe-mxd4	125	E	oe-meh	167	O	MAD46	209	I	PLLBYP	251	O	DATA58
42	I	MAD06	84	E	oe-mxd5	126	I	—	168	I	MAD47	210	O	IRL0	252	I	DATA57

I – Input    O – Output    E – Enable

Table 21-5. MXCC JTAG Boundary Scan Bit Order for MBus (Continued)

253	O	DATA57	294	I	DATA38	335	I	ADDR20	376	I	ADDR01	417	E	oe-we1	458	I	DATA18
254	I	DATA56	295	O	DATA38	336	O	ADDR20	377	O	ADDR01	418	I	DATA00	459	O	DATA19
255	O	DATA56	296	I	DATA37	337	E	oe-addr2	378	I	ADDR00	419	O	DATA00	460	I	DATA20
256	E	oe-data7	297	O	DATA37	338	I	ADDR19	379	O	ADDR00	420	I	DATA01	461	O	DATA20
257	E	oe-data6	298	I	DATA36	339	O	ADDR19	380	I	OE	421	O	DATA01	462	I	DATA21
258	I	DATA55	299	O	DATA36	340	I	ADDR18	381	O	OE	422	I	DATA02	463	O	DATA21
259	O	DATA55	300	I	DATA35	341	O	ADDR18	382	E	oe-oe	423	O	DATA02	464	I	DATA22
260	I	DATA54	301	O	DATA35	342	I	ADDR17	383	I	WR	424	I	DATA03	465	O	DATA22
261	O	DATA54	302	I	DATA34	343	O	ADDR17	384	O	WR	425	O	DATA03	466	I	DATA23
262	I	DATA53	303	O	DATA34	344	I	ADDR16	385	E	oe-wr	426	I	DATA04	467	O	DATA23
263	O	DATA53	304	I	DATA33	345	O	ADDR16	386	I	RD	427	O	DATA04	468	E	oe-data2
264	I	DATA52	305	O	DATA33	346	I	ADDR15	387	I	BURST	428	I	DATA05	469	E	oe-data3
265	O	DATA52	306	I	DATA32	347	O	ADDR15	388	O	RETRY	429	O	DATA05	470	I	DATA24
266	I	DATA51	307	O	DATA32	348	I	ADDR14	389	O	PEND	430	I	DATA06	471	O	DATA24
267	O	DATA51	308	O	WE3	349	O	ADDR14	390	O	MEXC	431	O	DATA06	472	I	DATA25
268	I	DATA50	309	O	WE2	350	I	ADDR13	391	O	WRDY	432	I	DATA07	473	O	DATA25
269	O	DATA50	310	O	WE1	351	O	ADDR13	392	O	RRDY	433	O	DATA07	474	I	DATA26
270	I	DATA49	311	O	WE0	352	I	ADDR12	393	O	WGRT	434	E	oe-data0	475	O	DATA26
271	O	DATA49	312	E	oe-we0	353	O	ADDR12	394	O	RGRT	435	E	oe-data1	476	I	DATA27
272	I	DATA48	313	E	oe-dpar1	354	E	oe-addr1	395	I	CMDS	436	I	DATA08	477	O	DATA27
273	O	DATA48	314	I	DPAR3	355	I	ADDR11	396	O	CMDS	437	O	DATA08	478	I	DATA28
274	I	DATA47	315	O	DPAR3	356	O	ADDR11	397	E	oe-cmds	438	I	DATA09	479	O	DATA28
275	O	DATA47	316	I	DPAR2	357	I	ADDR10	398	I	DEMAP	439	O	DATA09	480	I	DATA29
276	I	DATA46	317	O	DPAR2	358	O	ADDR10	399	O	DEMAP	440	I	DATA10	481	O	DATA29
277	O	DATA46	318	I	DPAR1	359	I	ADDR09	400	E	oe-dmap	441	O	DATA10	482	I	DATA30
278	I	DATA45	319	O	DPAR1	360	O	ADDR09	401	I	CSA	442	I	DATA11	483	O	DATA30
279	O	DATA45	320	I	DPAR0	361	I	ADDR08	402	I	LDST	443	O	DATA11	484	I	DATA31
280	I	DATA44	321	O	DPAR0	362	O	ADDR08	403	I	SYNC	444	I	DATA12	485	O	DATA31
281	O	DATA44	322	O	RESET	363	I	ADDR07	404	E	oe-dpar0	445	O	DATA12			
282	I	DATA43	323	O	WEE	364	O	ADDR07	405	I	DPAR4	446	I	DATA13			
283	O	DATA43	324	I	SIZE1	365	I	ADDR06	406	O	DPAR4	447	O	DATA13			
284	I	DATA42	325	I	SIZE0	366	O	ADDR06	407	I	DPAR5	448	I	DATA14			
285	O	DATA42	326	I	ERROR	367	I	ADDR05	408	O	DPAR5	449	O	DATA14			
286	I	DATA41	327	I	CCHBL	368	O	ADDR05	409	I	DPAR6	450	I	DATA15			
287	O	DATA41	328	I	SU	369	I	ADDR04	410	O	DPAR6	451	O	DATA15			
288	I	DATA40	329	I	ADDR23	370	O	ADDR04	411	I	DPAR7	452	I	DATA16			
289	O	DATA40	330	O	ADDR23	371	E	oe-addr0	412	O	DPAR7	453	O	DATA16			
290	E	oe-data5	331	I	ADDR22	372	I	ADDR03	413	O	WE4	454	I	DATA17			
291	E	oe-data4	332	O	ADDR22	373	O	ADDR03	414	O	WE5	455	O	DATA17			
292	I	DATA39	333	I	ADDR21	374	I	ADDR02	415	O	WE6	456	I	DATA18			
293	O	DATA39	334	O	ADDR21	375	O	ADDR02	416	O	WE7	457	O	DATA18			

I - Input O - Output E - Enable

**Note:**

Table 21-5 reflects MXCC Rev. 2.X BSCAN chain. MXCC Rev. 1.X BSCAN does not contain the MX and GTLREF1 bits. Future revisions may have different BSCAN chains.

Table 21-6.MXCC JTAG Boundary Scan Bit Order on XBus

1	I	MBSEL	43	O	XDATA06	85	I	XDATA24	127	I	XREQ2[1]	169	O	XDATA47	211	O	IRL1
2	I	LDATA7	44	I	XDATA07	86	O	XDATA24	128	I	XREQ3[0]	170	E	oemxd10	212	O	IRL2
3	O	LDATA7	45	O	XDATA07	87	I	XDATA25	129	I	XREQ3[1]	171	E	oemxd11	213	O	IRL3
4	I	LDATA6	46	E	oe-mxd0	88	O	XDATA25	130	O	XGNT0	172	I	XDATA48	214	I	ADDR24
5	O	LDATA6	47	E	oe-mxd1	89	I	XDATA26	131	I	XGNT1	173	O	XDATA48	215	O	ADDR24
6	I	LDATA5	48	I	XDATA08	90	O	XDATA26	132	O	XGNT1	174	I	XDATA49	216	I	ADDR25
7	O	LDATA5	49	O	XDATA08	91	I	XDATA27	133	I	XGNT2	175	O	XDATA49	217	O	ADDR25
8	I	LDATA4	50	I	XDATA09	92	O	XDATA27	134	O	XGNT2	176	I	XDATA50	218	I	ADDR26
9	O	LDATA4	51	I	QTLREF1	93	I	XDATA28	135	O	XGNT3	177	O	XDATA50	219	O	ADDR26
10	I	LDATA3	52	O	XDATA09	94	O	XDATA28	136	I	XDATA32	178	I	XDATA51	220	I	ADDR27
11	O	LDATA3	53	I	XDATA10	95	I	XDATA29	137	O	XDATA32	179	O	XDATA51	221	O	ADDR27
12	I	LDATA2	54	O	XDATA10	96	O	XDATA29	138	I	XDATA33	180	I	XDATA52	222	I	ADDR28
13	O	LDATA2	55	I	XDATA11	97	I	XDATA30	139	O	XDATA33	181	O	XDATA52	223	O	ADDR28
14	I	LDATA1	56	O	XDATA11	98	O	XDATA30	140	I	XDATA34	182	I	XDATA53	224	I	ADDR29
15	O	LDATA1	57	I	XDATA12	99	I	XDATA31	141	O	XDATA34	183	O	XDATA53	225	O	ADDR29
16	I	LDATA0	58	O	XDATA12	100	O	XDATA31	142	I	XDATA35	184	I	XDATA54	226	I	ADDR30
17	O	LDATA0	59	I	XDATA13	101	E	oe-mxd6	143	O	XDATA35	185	O	XDATA54	227	O	ADDR30
18	E	oe-bb-cl	60	O	XDATA13	102	E	oe-mxd7	144	I	XDATA36	186	I	XDATA55	228	E	oeaddr3
19	O	LCMDS	61	I	XDATA14	103	I	XPAR0	145	O	XDATA36	187	O	XDATA55	229	I	ADDR31
20	O	LCMD2	62	O	XDATA14	104	O	XPAR0	146	I	XDATA37	188	E	oemxd12	230	O	ADDR31
21	I	LCMD1	63	I	XDATA15	105	E	oe-merr	147	O	XDATA37	189	E	oemxd13	231	I	ADDR32
22	O	LCMD1	64	O	XDATA15	106	I	XPAR1	148	I	XDATA38	190	I	XDATA56	232	O	ADDR32
23	I	LCMD0	65	E	oe-mxd2	107	O	XPAR1	148	O	XDATA38	191	O	XDATA56	233	I	ADDR33
24	O	LCMD0	66	E	oe-mxd3	108	E	oe-mrdy	150	I	XDATA39	192	I	XDATA57	234	O	ADDR33
25	O	CCERR	67	I	XDATA16	109	I	XPAR2	151	O	XDATA39	193	O	XDATA57	235	I	ADDR34
26	E	oe-merr	68	O	XDATA16	110	O	XPAR2	152	E	oe-mxd6	194	I	XDATA58	236	O	ADDR34
27	I	spare-in	69	I	XDATA17	111	I	XPAR3	153	E	oe-mxd9	195	O	XDATA58	237	I	ADDR35
28	O	spare-out	70	O	XDATA17	112	O	XPAR3	154	I	XDATA40	196	I	XDATA59	238	O	ADDR35
29	I	RSTIN	71	I	XDATA18	113	E	oe-xpar	155	O	XDATA40	197	O	XDATA59	239	I	PCLK
30	I	XDATA00	72	O	XDATA18	114	I	XREQ0[0]	156	I	XDATA41	198	I	XDATA60	240	I	DATA63
31	O	XDATA00	73	I	XDATA19	115	O	XREQ0[0]	157	O	XDATA41	199	O	XDATA60	241	O	DATA63
32	I	XDATA01	74	O	XDATA19	116	E	oe-mbb	158	I	XDATA42	200	I	XDATA61	242	I	DATA62
33	O	XDATA01	75	I	XDATA20	117	I	XREQ0[1]	159	O	XDATA42	201	O	XDATA61	243	O	DATA62
34	I	XDATA02	76	O	XDATA20	118	O	XREQ0[1]	160	I	XDATA43	202	I	XDATA62	244	I	DATA61
35	O	XDATA02	77	I	XDATA21	119	E	oe-mas	161	O	XDATA43	203	O	XDATA62	245	O	DATA61
36	I	XDATA03	78	O	XDATA21	120	I	XREQ1[0]	162	I	XDATA44	204	I	XDATA63	246	I	DATA60
37	O	XDATA03	79	I	XDATA22	121	O	XREQ1[0]	163	O	XDATA44	205	O	XDATA63	247	O	DATA60
38	I	XDATA04	80	O	XDATA22	122	E	oe-mih	164	I	XDATA45	206	E	oemxd14	248	I	DATA59
39	O	XDATA04	81	I	XDATA23	123	I	XREQ1[1]	165	O	XDATA45	207	E	oemxd15	249	O	DATA59
40	I	XDATA05	82	O	XDATA23	124	O	XREQ1[1]	166	I	XDATA46	208	I	BCLK	250	I	DATA58
41	O	XDATA05	83	E	oe-mxd4	125	E	oe-msh	167	O	XDATA46	209	I	PLLBYP	251	O	DATA58
42	I	XDATA06	84	E	oe-mxd5	126	I	XREQ2[0]	168	I	XDATA47	210	O	IRL0	252	I	DATA57

I - Input O - Output E - Enable

Table 21–6. MXCC JTAG Boundary Scan Bit Order on XBus (Continued)

253	O	DATA57	294	I	DATA38	335	I	ADDR20	376	I	ADDR01	417	E	oe-we1	458	I	DATA19
254	I	DATA56	295	O	DATA38	336	O	ADDR20	377	O	ADDR01	418	I	DATA00	459	O	DATA19
255	O	DATA56	296	I	DATA37	337	E	oe-addr2	378	I	ADDR00	419	O	DATA00	460	I	DATA20
256	E	oe-data7	297	O	DATA37	338	I	ADDR19	379	O	ADDR00	420	I	DATA01	461	O	DATA20
257	E	oe-data6	298	I	DATA36	339	O	ADDR19	380	I	OE	421	O	DATA01	462	I	DATA21
258	I	DATA55	299	O	DATA36	340	I	ADDR18	381	O	OE	422	I	DATA02	463	O	DATA21
259	O	DATA55	300	I	DATA35	341	O	ADDR18	382	E	oe-oe	423	O	DATA02	464	I	DATA22
260	I	DATA54	301	O	DATA35	342	I	ADDR17	383	I	WR	424	I	DATA03	465	O	DATA22
261	O	DATA54	302	I	DATA34	343	O	ADDR17	384	O	WR	425	O	DATA03	466	I	DATA23
262	I	DATA53	303	O	DATA34	344	I	ADDR16	385	E	oe-wr	426	I	DATA04	467	O	DATA23
263	O	DATA53	304	I	DATA33	345	O	ADDR16	386	I	RD	427	O	DATA04	468	E	oe-data2
264	I	DATA52	305	O	DATA33	346	I	ADDR15	387	I	BURST	428	I	DATA05	469	E	oe-data3
265	O	DATA52	306	I	DATA32	347	O	ADDR15	388	O	RETRY	429	O	DATA05	470	I	DATA24
266	I	DATA51	307	O	DATA32	348	I	ADDR14	389	O	PEND	430	I	DATA06	471	O	DATA24
267	O	DATA51	308	O	WE3	349	O	ADDR14	390	O	MEXC	431	O	DATA06	472	I	DATA25
268	I	DATA50	309	O	WE2	350	I	ADDR13	391	O	WRDY	432	I	DATA07	473	O	DATA25
269	O	DATA50	310	O	WE1	351	O	ADDR13	392	O	RRDY	433	O	DATA07	474	I	DATA26
270	I	DATA49	311	O	WE0	352	I	ADDR12	393	O	WGRT	434	E	oe-data0	475	O	DATA26
271	O	DATA49	312	E	oe-we0	353	O	ADDR12	394	O	RGRT	435	E	oe-data1	476	I	DATA27
272	I	DATA48	313	E	oe-dpar1	354	E	oe-addr1	395	I	CMDS	436	I	DATA08	477	O	DATA27
273	O	DATA48	314	I	DPAR3	355	I	ADDR11	396	O	CMDS	437	O	DATA08	478	I	DATA28
274	I	DATA47	315	O	DPAR3	356	O	ADDR11	397	E	oe-cmds	438	I	DATA09	479	O	DATA28
275	O	DATA47	316	I	DPAR2	357	I	ADDR10	398	I	DEMAP	439	O	DATA09	480	I	DATA28
276	I	DATA46	317	O	DPAR2	358	O	ADDR10	399	O	DEMAP	440	I	DATA10	481	O	DATA28
277	O	DATA46	318	I	DPAR1	359	I	ADDR09	400	E	oe-demap	441	O	DATA10	482	I	DATA30
278	I	DATA45	319	O	DPAR1	360	O	ADDR09	401	I	CSA	442	I	DATA11	483	O	DATA30
279	O	DATA45	320	I	DPAR0	361	I	ADDR08	402	I	LDST	443	O	DATA11	484	I	DATA31
280	I	DATA44	321	O	DPAR0	362	O	ADDR08	403	I	SYNC	444	I	DATA12	485	O	DATA31
281	O	DATA44	322	O	RESET	363	I	ADDR07	404	E	oe-dpar0	445	O	DATA12			
282	I	DATA43	323	O	WEE	364	O	ADDR07	405	I	DPAR4	446	I	DATA13			
283	O	DATA43	324	I	SIZE1	365	I	ADDR06	406	O	DPAR4	447	O	DATA13			
284	I	DATA42	325	I	SIZE0	366	O	ADDR06	407	I	DPAR5	448	I	DATA14			
285	O	DATA42	326	I	ERROR	367	I	ADDR05	408	O	DPAR5	449	O	DATA14			
286	I	DATA41	327	I	CCHBL	368	O	ADDR05	409	I	DPAR6	450	I	DATA15			
287	O	DATA41	328	I	SU	369	I	ADDR04	410	O	DPAR6	451	O	DATA15			
288	I	DATA40	329	I	ADDR23	370	O	ADDR04	411	I	DPAR7	452	I	DATA16			
289	O	DATA40	330	O	ADDR23	371	E	oe-addr0	412	O	DPAR7	453	O	DATA16			
290	E	oe-data5	331	I	ADDR22	372	I	ADDR03	413	O	WE4	454	I	DATA17			
291	E	oe-data4	332	O	ADDR22	373	O	ADDR03	414	O	WE5	455	O	DATA17			
292	I	DATA39	333	I	ADDR21	374	I	ADDR02	415	O	WE6	456	I	DATA18			
293	O	DATA39	334	O	ADDR21	375	O	ADDR02	416	O	WE7	457	O	DATA18			

I – Input O – Output E – Enable

**Note:**

Table 21–6 reflects MXCC Rev. 2.X BSCAN chain. MXCC Rev. 1.X BSCAN does not contain the MX and GTLREF1 bits. Future revisions may have different BSCAN chains.

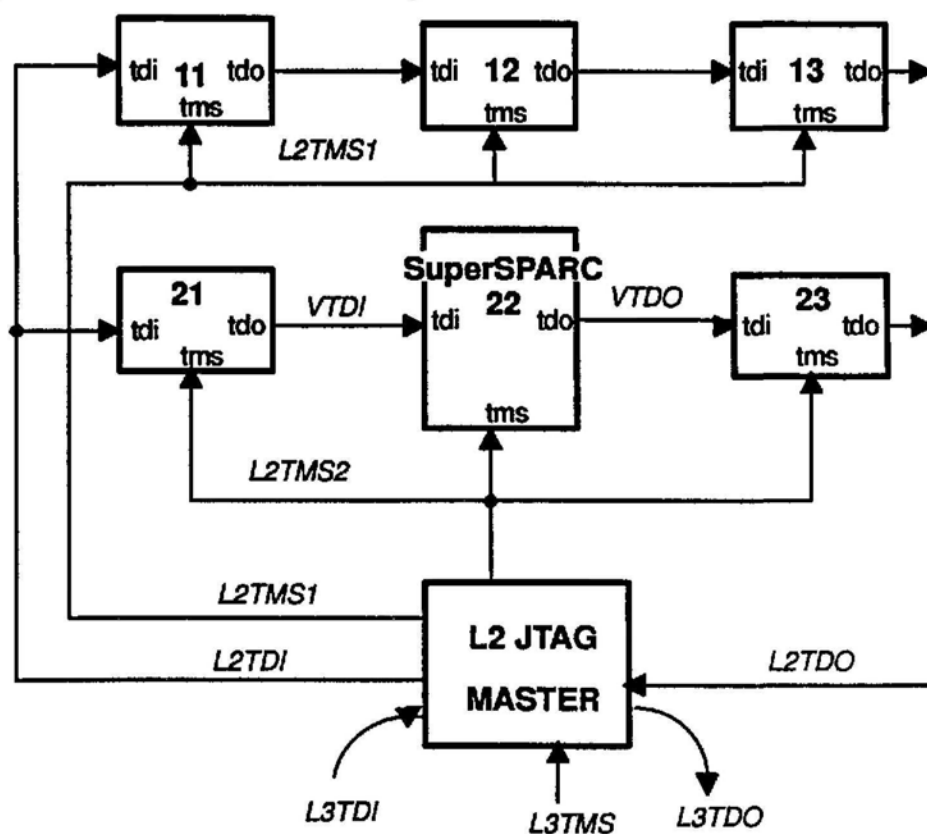


## 21.7 System-Level Test

The *IEEE 1149.1 Standard Access Port And Boundary Scan Architecture Specification* section, "Legal Interconnections Of Components Compatible With 1149.1" provides guidelines on how to connect TCK, TMS, TDI, and TDO for making serial, parallel, and hierarchical configurations for JTAG board level sub-systems. This allows a hierarchical JTAG test technique that SuperSPARC supports. Similar in ways to how SuperSPARC selects TDR scan chains, a second-level JTAG controller (e.g., board-level JTAG busmaster) may connect multiple chips on a board to form parallel and serial paths. Furthermore, a third-level JTAG controller (e.g., backplane-level JTAG busmaster) may connect multiple second-level JTAG controllers to form a chain.

Example 21-1 shows one board-level JTAG busmaster controlling six JTAG components configured into two parallel daisy chains. Each component in the chain connects its TDI to the preceding chip's TDO. The first chip in each of the parallel daisy chains is tied to a common level-two TDI, and the last chip in each chain is wire-ORed to a common level-two TDO. Each of the parallel chains receives a unique TMS from the second-level TAP controller.

Example 21-1. System-Level JTAG Test Hierarchy



## **Scan-Based Debug**

---

The SuperSPARC processor (SSP) uses the JTAG 1149.1 serial scan interface to allow access to scan-based debug features. These scan-based debug features allow you to debug systems in a non-intrusive manner.

<b>Topic</b>	<b>Page</b>
<b>22.1 Scan-Based Debug Support</b> .....	<b>22-2</b>
<b>22.2 The Scan-Based Debugger</b> .....	<b>22-3</b>
<b>22.3 Scan Registers</b> .....	<b>22-4</b>
<b>22.4 Scan-Based Debug Register In ASI Space</b> .....	<b>22-12</b>
<b>22.5 Entering Scan-Based Debug Mode</b> .....	<b>22-14</b>
<b>22.6 Scan-Based Debug Operation</b> .....	<b>22-15</b>
<b>22.7 Approximate Latencies for Each SDB Primitive</b> .....	<b>22-29</b>



## **22.1 Scan-Based Debug Support**

The SSP provides facilities to observe and control processor execution from a remote device using the IEEE 1149.1 JTAG serial scan interface. The JTAG interface is described in chapter 21.

Traditionally, systems debugging methods required very expensive dedicated add-on hardware, connected using ribbon cables and fragile connectors, and were not generally available when the first processor prototypes were delivered (which is when they would have been most useful). Furthermore, the length of the cable limited the processor system clock rate when the emulator was being used. This also introduced extra electrical loading, which affected pin timings.

As a solution to that problem, SuperSPARC provides scan-based debug logic to support a remote environment for debugging systems, done entirely over the serial JTAG bus. The features are useful for both hardware and software development. It is completely non-intrusive into the system design. Neither the pin timings nor the processor speed is affected. Nearly all features of traditional emulators are provided, except for real-time trace and memory emulation.

All programmer-visible state is accessible and changeable using the JTAG interface. Software must be provided to control the scan-based debug from a remote computer with a JTAG interface. During scan-based debug, the caches and store buffer continue to operate; these resources will snoop incoming system bus requests. Many scan-based debug resources are shared with standard software-debugging features.

### ***Scan-Based Debug Strategy***

SuperSPARC provides scan-based debug logic to aid in system debug and failure analysis. The scan-based debugger's (SDB) interface to SuperSPARC uses the JTAG instruction register (IR) to select one of four scan-based debug JTAG Test Data Registers (TDR). The SDB provides SuperSPARC with protocol commands, SDB instructions, addresses, and data for updating SuperSPARC state through the Scan Command and Instruction (MCI) TDR and Scan Data In (MDIN) TDR. SuperSPARC returns existing system state data and protocol status through two other JTAG TDRs: Scan Data Out (MDOUT) and Scan STATUS (MSTAT).

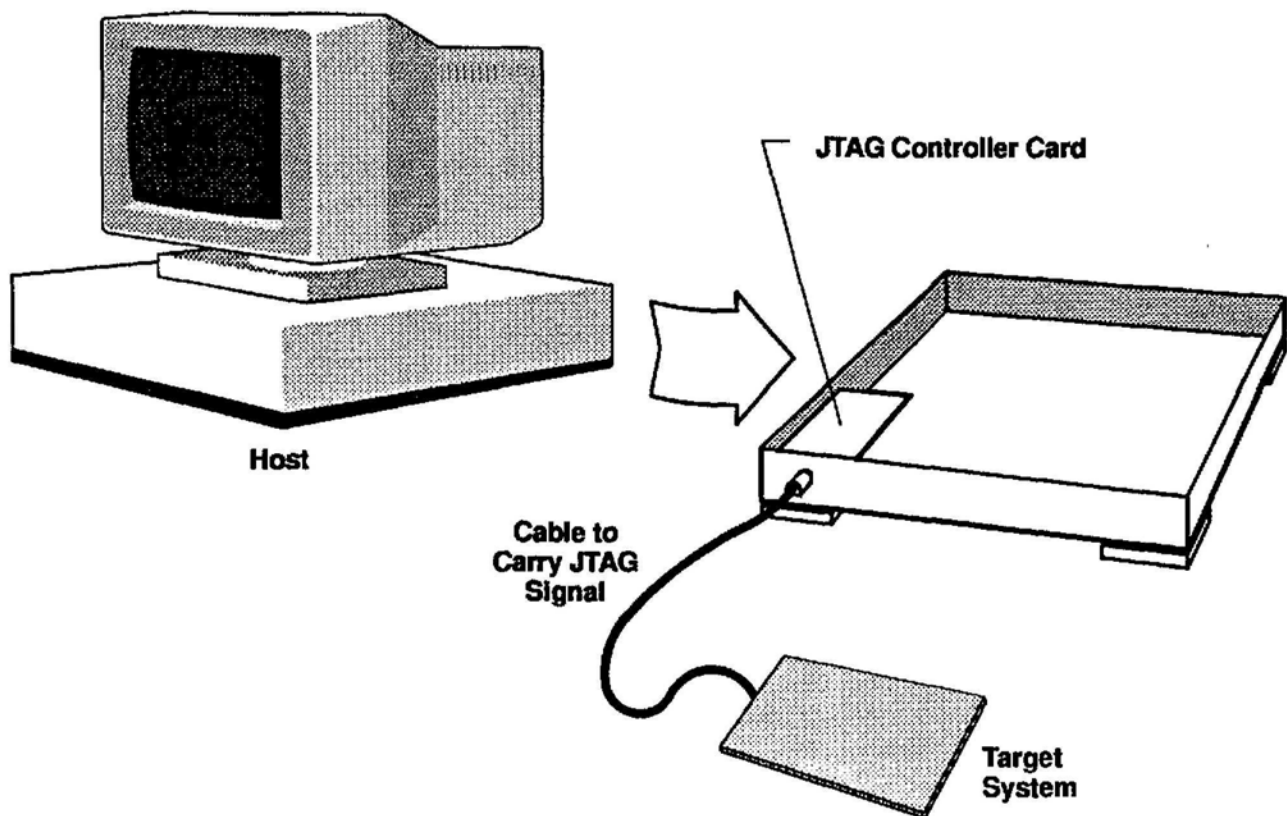
Through these registers, the SDB can command the processor to temporarily halt execution of the normal SPARC instruction stream. Once halted, the processor can be directed to execute any normal SPARC instruction. No processor state information not explicitly modified by this scan-based debug will be altered.

## 22.2 The Scan-Based Debugger

The SDB allows us to run scan-debug primitives on the SSP. The SDB consists of hardware and software along with a user interface to allow us to enter scan-based debug mode and scan instructions into the MCI register and check the status of the instruction execution.

The hardware portion of the SDB consists of a host system, a JTAG controller card, and a cable to carry the JTAG signals to the target system. The JTAG controller card must be capable of receiving commands from the host system and converting them into appropriate serial JTAG signals. It must also take the serial JTAG signals from the target system under test and convert them into a format that the host system can understand. See Figure 22-1 for the system hardware configuration.

*Figure 22-1. SDB Hardware Configuration*



The SDB software should, as a minimum, contain a driver routine to drive the JTAG controller card and higher-level software to allow you to easily write programs to run on the target system while in scan-based debug mode.

## 22.3 Scan Registers

There are five JTAG TDRs that are important for scan-based debug; they are listed in Table 22-1.

Table 22-1. JTAG TDR Scan Registers

JTAG TDR	Name
IR	Instruction Register
MCI	Scan Command and Instruction
MDIN	Scan Data In
MDOUT	Scan Data OUT
MSTAT	Scan STATUS

See chapter 21 for details on JTAG TDR scan operation.

### 22.3.1 Instruction Register (IR)

The IR register selects which JTAG TDR scan chain to access. The following table only lists IR encoding that selects scan register.

Table 22-2. Scan Register Selection

Register Selected	IR Value	# bits
MCI	0x08	37
MDIN	0x09	32
MDOUT	0x0a	32
MSTAT	0x0b	13

### 22.3.2 Scan Data In (MDIN)

The MDIN register is a 32-bit register that allows information to be passed from the scan controller to the SSP. The format of the MDIN register is shown in Table 22-3.

Table 22-3. MDIN Register Format



Data is scanned into the MDIN register from the SDB. An instruction sequence can move this data from the JTAG MDIN register into the SSP integer register file using a load instruction and a SPARC address space identifier (ASI) access. The ASI is an eight-bit value that is appended to the address of a memory access. The purpose of the ASI is to identify special modes and address spaces. The ASI for access to the MDIN register is 0x44. The specific SPARC instruction that will perform this operation is:

LDA [%g0] 0x44 , %reg

where %reg is an integer register. Subsequent scan debug instructions can use this data to update memory or processor state.

### 22.3.3 Scan Command and Instruction (MCI)

The MCI scan chain is 37 bits and comprises two fields: a five-bit scan command register (MCMD) and a 32-bit scan instruction register (MINST). The format of the MCI scan register is shown in Table 22-4.

Table 22-4. Scan Command and Instruction Register (MCI)

MCMD	MINST
36	31 0

**MCMD** Scan Command Register. Its component fields are shown in Table 22-5.

**MINST** Scan Instruction. This register contains a single SPARC instruction that is executed as the scan instruction (qualified by MEXEC). Several scan instructions are typically required to completely execute the semantics of a scan primitive.

The MCMD field contains the information that qualifies the 32-bit SPARC instruction in the MINST field. All bits in the MCMD field are cleared when the JTAG TAP controller resets.

Table 22-5. Scan Command Register (MCMD)

INITM	MENTER	MEXEC	MEXIT	MRESET
36	35	34	33	32

<b>INITM</b>	Enable Scan-Based Debug. The primary function of this bit is to enable SuperSPARC to enter scan-based debug as a result of a breakpoint, if the ACTION register is properly programmed. In addition, this bit affects the operation of signal scan-based debug (SIGM), a SuperSPARC-specific instruction. If INITM=0, SIGM will execute as a NOP. If INITM=1, SIGM initiates a user-level scan-based debug mode entry. This bit is cleared on JTAG TAP controller reset.
<b>MENTER</b>	Enter scan-based debug mode. When set, SuperSPARC is forced to enter scan-based debug mode. The program counter (PC/NPC pair) is captured to resume execution after scan-based debug exit. In scan-based debug mode, the SuperSPARC prefetch controller stops accessing the instruction cache, starts passing NOPs into the IU pipeline, and examines the state of MEXEC to wait for an instruction to execute. This bit is cleared on JTAG TAP controller reset.
<b>MEXEC</b>	Execute MINST. When set, a single instance of the instruction in the MINST register will be forced into the processor pipeline. This will cause it to be executed as a normal SPARC instruction. Once launched, the prefetch controller will clear MEXEC, resume passing NOPs into the IU pipeline, and monitor valid bits at the last stage of the IU pipeline. Once the prefetch controller determines that no remaining instructions are in the processor pipeline, it examines MEXIT to determine whether to remain in scan-based debug mode or to resume normal execution. The MEXEC bit is cleared on JTAG TAP controller reset.

**MEXIT**

Exit scan-based debug mode. When set, SuperSPARC will exit scan-based debug mode (to resume normal execution) as soon as all execution in the pipeline is complete. The execution stream branches to the PC/NPC values stored on entry to scan-based debug mode. The prefetch controller continues to pass NOP into the pipeline until either an MEXEC or MEXIT is asserted. This bit is cleared on JTAG TAP controller reset.

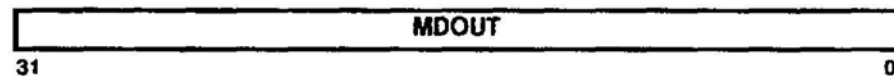
**MRESET**

Hardware Reset. When set, this bit forces a full SuperSPARC hardware reset (rather than a watchdog reset). This bit is cleared on JTAG TAP controller reset.

**22.3.4 Emulation Data Out (MDOUT)**

The MDOUT register stores data to be passed back to the scan controller card. The format of the MDOUT register is shown in Table 22-6.

*Table 22-6. Scan Data Out Register (MDOUT)*



An instruction sequence can move outgoing processor state data from the integer register file to the MDOUT register by storing a word to the 32-bit ASI space for the MDOUT register. The special ASI to access MDOUT is 0x46. The specific instruction that performs this operation is:

STA %reg, [%g0] 0x46

where %reg is a SPARC integer register. The SDB can then scan this data out of MDOUT and display the information or status.

### 22.3.5 Scan Status (MSTAT)

The MSTAT scan chain is 13 bits long, and it contains information about SuperSPARC scan debug status. The only way to retrieve this information is through JTAG scan operation. The MSTAT register needs to be polled after every instruction to ensure that the instruction completed execution and that no error occurred. The MSTAT register is cleared upon TAP Controller reset. The MSTAT format is:

ECHOTMR	MACK	TMRM	CBKM	ZICM	DBKM	ZCCM
12	11	10	9	8	7	6

IPND	ERRMODE	MIFLTD	PFPX	MIDONE	FQE
5	4	3	2	1	0

#### ECHOTMR

Echoed MCMD.MENTER. ECHOTMR is an echoed version of MCMD.MENTER after it has passed through TCK-VCLK-TCK synchronization. (TCK is test clock, VCLK is SuperSPARC clock). ECHOTMR is asserted asynchronously to scan instruction execution. The purpose of this signal is to signal that SuperSPARC has seen the request to enter scan-based mode. Assertion of this signal does not indicate that SuperSPARC has actually entered scan-based debug mode. This bit is cleared upon the JTAG TAP Controller reset.

#### MACK

Scan-Based Debug Acknowledge. MACK is an indication that SuperSPARC is in scan-based debug mode. It is asserted as soon as SuperSPARC enters scan-based debug mode and stays active until SuperSPARC leaves scan-based debug mode. It is synchronized to TCK (refer to chapter 21 – JTAG). This bit is cleared upon TAP controller reset.

#### Note:

TMRM, CBKM, ZICM, DBKM, and ZCCM qualify MACK to identify to the SDB the cause for SuperSPARC's entry into scan-based debug mode. It is possible that more than one is asserted, indicating that there was more than one cause.

<b>TMRM</b>	Entered scan-based debug mode from MENTER request. TMRM indicates that SuperSPARC entered scan-based debug mode due to an MCMD.MENTER request from the SDB. This bit is cleared on JTAG TAP controller reset or SuperSPARC hardware reset, or by updating the MCI register.
<b>CBKM</b>	Code Address Breakpoint. CBKM indicates that SuperSPARC entered scan-based debug mode due to a code address breakpoint. (See Chapter 15.) This bit is cleared on JTAG TAP controller reset, SuperSPARC hardware reset, or by updating the MCI register.
<b>ZICM</b>	Zero Instruction Count Breakpoint. ZICM indicates that SuperSPARC entered scan-based debug mode due to a zero-instruction-count breakpoint. (See Chapter 15.) This bit is cleared on JTAG TAP Controller reset, SuperSPARC reset, or by updating the MCI register.
<b>DBKM</b>	Data Address Breakpoint. DBKM indicates that SuperSPARC entered scan-based debug mode due to a data-address breakpoint. (See Chapter 15.) This bit is cleared on JTAG TAP controller reset, SuperSPARC reset, or by updating the MCI register.
<b>ZCCM</b>	Zero Cycle Count. ZCCM indicates that SuperSPARC entered scan-based debug mode due to a zero cycle count breakpoint. (See Chapter 15.) This bit is cleared upon a JTAG TAP controller reset, SuperSPARC reset, or by updating the MCI register.
<b>IPND</b>	Pending Interrupt. IPND assertion indicates that the processor has a pending interrupt request that is higher than the current SuperSPARC PSR.IPL or at level 15. IPND is used to inform the SDB that an interrupt is pending and that SuperSPARC should be released to service it. This bit is cleared upon the JTAG TAP controller reset.



## **ERRMODE**

Error Mode. ERRMODE indicates SuperSPARC has entered error mode as a result of a fault generated while in scan-based debug mode. Any exception occurring during a scan-based debug instruction sequence will force SuperSPARC into error mode. Entering error mode will induce the watchdog reset sequence, set MSTAT.ERRMODE, and exit scan-based debug mode. The ERRMODE bit will stay asserted until the next MSTAT update operation or is cleared by the JTAG TAP controller reset.

---

### **Note:**

When error mode occurs while in scan-based debug mode, no assertion of MEXIT or MRESET is needed for SuperSPARC to leave scan-based debug mode and restart execution at the reset vector.

---

## **MIFLTD**

Data Access Fault. MIFLTD indicates that a scan-based debug instruction with a data memory reference created a data access fault. It does not indicate whether other sources of exceptions (e.g., pending FP exceptions) occurred during scan-based debug mode instruction execution. MIFLTD is meaningful only when qualified by the assertion of MIDONE status bit. There is no scan-based debug instruction exception update to MFSR or MFAR. Only MSFSR (the shadow FSR) will be updated. MSFSR is cleared on entrance to scan-based debug mode (see Chapter 10 for more details). The SDB service processor must check the MIFLTD status bit after every scan-based debug instruction that references memory, and, if that bit is set, it is the responsibility of the SDB to clear it with an explicit STA instruction to the MSFSR, a JTAG TAP controller reset, by updating MCI, or by a SuperSPARC reset.

## **PFPX**

Pending Floating-Point Exception. PFPX indicates that a pending floating-point exception exists. This exception can be caused in two ways: prior non-scan-based debug FPOPs (which were already in the FQ and continued to execute while the processor was in scan-based debug mode) generate an FP exception. Scan-based debug FPOPs can also generate an FP exception.

**Note:**

Before issuing a floating-point related scan-based debug instruction, the SDB must make sure that all FPOPs have cleared from the FQ (MSTAT.FQE = 1) and that there is no pending FP exception generated (PFPX = 0).

Any taken floating-point exception in scan-based debug mode will cause error mode. The PFPX bit is cleared by clearing out the floating-point queue (FQ), by a SuperSPARC hardware reset, or by a JTAG TAP controller reset.

**MIDONE**

Scan-Based Debug Instruction Completed. MIDONE is asserted when a scan-based debug instruction completes execution. No additional scan-based debug instructions should be issued until the current scan-based debug instruction has completed, as indicated by MIDONE. If a scan-based debug instruction is an FPOP, MIDONE assertion only means that the FPOP was successfully issued to the FPU. Therefore, for floating-point-related scan-based debug instructions, the requirements to satisfy before issuing another FP-related scan-based debug instruction is error-free execution of the previous FPOP, which is indicated when FQE = 1 and PFPX = 0. This MIDONE bit is cleared by JTAG TAP controller reset, reading MSTAT, subsequent entry into scan-based debug mode, or SuperSPARC hardware reset.

**FQE**

FQ Empty. FQE indicates that the floating-point queue (FQ) is empty and is asserted when all FPOPs have finished error-free execution.

## 22.4 Scan-Based Debug Registers in ASI Space

There are four ASI-mapped registers that directly support scan-based debug: MPC/MNPC, MTMP1/ MTMP2, MDIN, and MDOUT. The following subsections briefly describe each of those registers.

### 22.4.1 Scan-Based Debug Exit PC/nPC Registers (MPC/MNPC)

ASI	Function	Access	Size
0x47	Scan-based debug Exit PC	LD/ST	single
0x48	Scan-based debug Exit nPC	LD/ST	single

The program counter of the instruction executing when scan-based debug mode entry occurs is written to PC/nPC registers. These values are the targets of the branch when SuperSPARC resumes normal execution upon exit from scan-based debug mode. These registers can be examined or altered by the SDB and are accessible through LDA/STA 0x47-0x48.

### 22.4.2 Scan Temporary Registers (MTMP1/MTMP2)

ASI	Function	Access	Size
0x40	Scan Temp 1	LD/ST	single
0x41	Scan Temp 2	LD/ST	single

The MTMP1 and MTMP2 registers are useful for temporarily storing information while in scan-based debug mode. If more than two words of information need to be temporarily stored, the SDB must use its JTAG MDOUT scan capability and later restore it through JTAG MDIN. The MTMP1 and MTMP2 registers are accessible through ASI 0x40 and 0x41.

### 22.4.3 Scan Data In (MDIN) Register

ASI	Function	Access	Size
0x44	Scan data in	LD	single

A scan-based debug instruction sequence can move incoming scan-based debug data from the JTAG MDIN register into the integer unit register file using an LDA instruction. Subsequent scan-based debug instructions can use this data to update memory or SuperSPARC state. The MDIN register is a read-only register through an ASI 0x44 access. See Subsection 22.3.2 for more details on MDIN.

#### **22.4.4 Scan Data Out (MDOUT) Register**

<b>ASI</b>	<b>Function</b>	<b>Access</b>	<b>Size</b>
0x46	Scan data out	ST	single

A scan-based debug instruction sequence can move outgoing SuperSPARC state data from the integer unit register file to the scan-based debug MDOUT register by using an STA. The SDB can then use this information to display processor state or check status. This register is accessible through ASI 0x46. See Subsection 22.3.4 for more details on MDOUT.

## **22.5 Entering Scan-Based Debug Mode**

Scan-based debug mode can be entered through any one of six different methods:

- ☐ Set MCMD.MENTER to force scan-based debug mode.
- ☐ Execute SIGM with MIMD.INTM set.
- ☐ Through Code Address Breakpoint.
- ☐ Through Data Address Breakpoint.
- ☐ Through Instruction Counter Breakpoint
- ☐ Through Cycle Counter Breakpoint.

The first is used by the SDB to unconditionally present a scan-based debug mode request by using the JTAG tap controller to assert MCMD.MENTER. When the MCMD.MENTER bit is set, the processor will enter scan-based debug mode on the next SPARC instruction. The other methods require ASI registers to be configured to conditionally generate a scan-based debug mode request. SIGM (a special SuperSPARC instruction) also allows scan-based debug mode entry. See Table 15-1 for further description on how to set up the breakpoints.

## 22.6 Scan-Based Debug Operation

### 22.6.1 Scan-Based Debug Execution Details

This subsection provides some additional information on how SuperSPARC operates in scan-based debug mode.

#### 22.6.1.1 State During Scan-Based Debug Mode

On entry to scan-based debug mode, a store buffer copyout is initiated. The MCNTL.SB bit is set to 0, thereby turning off the store buffer. All store instructions are synchronous and bypass the store buffer. In scan-based debug mode, all instructions execute in supervisor mode, and multiple instruction execution is disabled. The PSR.CWP, PSR.PS, and TBR.TT remain at the values they were before scan-based debug mode entry. The MCNTL.NF bit is asserted, and the PSR.ET bit is cleared. The MFSR and MFAR registers remain unaffected by scan-based debug faults. The fault information is written into the shadow MFSR and MFAR registers reserved exclusively for scan-based debug mode. The FPU remains enabled and will continue instruction execution without being aware that the processor has entered scan-based debug mode.

Table 22-7 lists the possible states that the SSP can be in while in scan-based debug mode.

Table 22-7. SuperSPARC State in Scan-Based Debug Mode

Register/Bit Affected	State
PSR.S	asserted
PSR.EF	asserted
PSR.ET	negated
ACTION.MIX	negated
MCNTL.NF	asserted
MCNTL.SB	negated

### 22.6.1.2 Exceptions During Scan-Based Debug Instruction Execution

The SSP executes instructions with PSR.ET disabled during scan-based debug mode. Any synchronous exceptions that are reported will cause the SSP to enter error mode and set MSTAT.ERRMODE, which will induce a watchdog reset. The fact that PSR.ET was disabled on entry to scan-based debug mode allows asynchronous exceptions, such as priority interrupts, data\_store\_exception, and floating\_point\_exception to be ignored.

If a scan-based debug data memory reference instruction faults, it sets the MSTAT.MIFLTD bit but does not enter error mode. The shadow FSR is set with the appropriate values to describe the cause of the fault. The SDB needs to read this register when the MSTAT.MIFLTD bit is set to determine the cause of the fault and also to clear out the shadow FSR. Failure to do this will leave MSTAT.MIFLTD asserted and can give subsequent scan-based debug instructions a faulty status indication.

MSTAT.IPND informs the SDB that a priority interrupt is being deferred while in scan-based debug mode. The SDB will need to determine whether to exit scan-based debug mode to allow the processor to deal with the priority interrupt.

MSTAT.PFPX informs the SDB that a priority interrupt is being deferred while in scan-based debug mode. Floating-point exceptions during scan-based debug mode are of particular concern, requiring elaborate trap handlers. Care must be taken to examine all the signals indicating error-free condition before issuing a floating-point-related instruction. See Section 22.3.

### 22.6.1.3 Legal and Illegal Scan-Based Debug Instructions

The scan-based debug instruction in MINST must be a legal SPARC instruction. Only a subset of the SPARC instruction set, however, is supported during scan-based debug mode. In general, instructions that affect the flow of execution when not in scan-based debug mode are not supported. A simple example is a branch instruction. There is no reason to support control transfer instructions in scan-based debug mode. Operation of the processor in scan-based debug mode on these illegal instructions is undefined.

Legal scan-based debug instructions:

- ☐ All legal memory reference instructions (including ASI accesses and atomic accesses).
- ☐ All arithmetic and logical instructions, except trapping tagged arithmetic.

- ☐ SETHI.
- ☐ All integer state register accesses.

Illegal scan-based debug instructions (but legal SPARC instructions):

- ☐ All control transfer instructions (CALL, Bicc, FBicc, JUMPL, and RETT).
- ☐ All software traps (Ticc).
- ☐ The FLUSH instruction.
- ☐ All trapping tagged arithmetic.
- ☐ SAVE and RESTORE (manipulate CWP directly instead).
- ☐ All illegal instructions.

Many of these illegal operations will cause entry into error mode, force SuperSPARC to leave scan-based debug mode, and induce a non-scan-based debug watchdog reset.

#### **22.6.1.4 Compound Scan-Based Debug Protocol Commands**

Separate control bits are used to enter scan-based debug mode, execute a scan-based debug instruction, and then exit. These separate bits may be used together to optimize scan-based debug sequences. The simplest scan-based debug sequence writes the MCI.MINST register with a single scan-based debug instruction and sets the MENTER, MEXEC, and MEXIT bits. This will cause SuperSPARC to enter scan-based debug mode, execute the scan-based debug instruction, and then resume execution. This entire sequence requires only a few cycles to execute after MCI is scanned in through the JTAG interface. Table 22-8 describes the valid compound scan-based debug sequences.



Table 22-8. Valid Compound Scan-Based Debug Mode Sequences

MENTER	MEEXEC	MEXIT	Action
1	1	0	Enter scan-based debug mode, issue a command, then pause awaiting another MEEXEC or MEXIT
x	1	0	Once in scan-based debug mode, issue a new scan-based debug mode instruction, then pause awaiting another MEEXEC or MEXIT.
x	1	1	Once in scan-based debug mode, issue a new scan-based debug instruction, exit scan-based debug mode upon completion, then resume execution at the captured (or altered) non-scan-based debug mode PC pair.
x	0	1	Once in scan-based debug mode, immediately resume execution at the captured (or altered) non-scan-based debug mode PC pair.
x	0	0	This is an NOP in scan-based debug mode.
1	0	0	Will cause entry into scan-based debug mode, and wait.
1	0	1	Will start entry into scan-based debug mode, but then immediately exits. No scan-based debug instruction is executed. MSTAT.MACK is not updated. To the programmer it might appear that scan-based debug mode was never entered. (This sequence is not very useful.)

Simultaneous assertion of MEEXEC and MEXIT can lead to hazardous timing races in sampling MSTAT.MIDONE that cause unfavorable interferences from the SDB. On completion of the last scan-based debug instruction in the current scan-based debug session, this compound MCMD mode will exit scan-based debug mode. MSTAT.MIDONE will be set, and MSTAT.MACK will be negated.

The SDB must check MSTAT to make sure the previous scan-based debug instruction has completed and was error-free. A re-entry into scan-based debug mode clears out MSTAT.MIDONE and MSTAT.MIFLTD, and it could do so before the SDB could check MSTAT. In this case, the SDB would mistakenly assume that the last scan-based debug instruction had not completed.

It is recommended that an MEEXEC be issued without MEXIT. An immediate re-entry into scan-based debug mode will allow the scan-based debug program to differentiate between the end of the first scan-based debug session and the start of the second scan-based debug session. When the scan-based debug instruction completes with no faults, an MEXIT can be issued.

## 22.6.2 Scan-Based Debug Sequences

Issuing each scan-based debug instruction requires the SDB to send multiple JTAG scan sequences. Since the MCI register contains both the MINST and MCMD registers, only one register needs to be loaded to issue a single scan-based debug instruction. Depending on the scan-based debug instruction to execute, MDIN may need to be set before the instruction is executed. For more complex scan-based debug sequences, processor state will need to be preserved before any state is modified. This involves many scan-based debug instructions.

Unless SuperSPARC is in error mode, the SDB can force SuperSPARC into scan-based debug mode by scanning in an asserted MCMD.MENTER value. The SDB polls MSTAT for an indication that the instruction is complete. The scan sequence for emulating an individual instruction is at least a portion of the following:

Scan-In any required pointers and data into MDIN.  
Scan-In the scan-based debug instruction (MCI\_MINST) and scan-based debug protocol command (MCMD), including the EXEC and optionally the MEXIT bits.  
Scan-Out MSTAT scan-based debug status register to determine when the scan-based debug instruction has completed, faulted, or induced error mode. This poll also indicates whether any prioritized interrupt is currently at SuperSPARC's pins.  
Scan-Out of any requested SuperSPARC system state data from MDOUT.

## 22.6.3 Scan-Based Debug Instruction Sequences for Common SDB Functions

This subsection will describe some possible scan-based debug instruction sequences for some common primitives. Many other implementations are possible. The primitives may be combined to create other functions as needed.

The primitives to be described are:

- ☐ Read/Write Integer Registers.
- ☐ Read/Write Integer Control Registers (PSR, WIM, TBR, and Y).
- ☐ Read/Write Floating-Point Registers.
- ☐ Read/Write Floating-Point Control Registers.
- ☐ Read/Write Memory (byte, half, word, double, etc.).
- ☐ Read/Write Memory (Normal and ASI).
- ☐ Set Code and Data Address Breakpoints.
- ☐ Single-Step.

- ☐ Run for N Cycles.
- ☐ Run until Breakpoint reached.

The next subsections provide detailed sequences for each of the above operations. Throughout these sequences, numerous symbolic constants will be used, which are described in Table 22-9. All sequences assume that the processor has entered scan-based debug mode without fault.

Table 22-9. Symbolic Constants for Scan-Based Debug Sequences

Constant	Actual
mdiag	0x38
bkb	0x000
blm	0x100
bkc	0x200
bks	0x300
mtmp1	0x40
mtmp2	0x41
mdin	0x44
mdout	0x46
mpc	0x47
mnp	0x48
ctrv	0x49
ctrc	0x4a
ctrs	0x4b
action	0x4c
XREG	Scan-based debug register
XADDR	Scan-based debug memory addr

### 22.6.3.1 Integer Register File Read

The most basic SDB operation is to read an integer register. The following scan-based debug instruction sequence will transfer the contents of the integer register XREG within the Current Window Pointer (CWP) to MDOUT register. Once in MDOUT, the data can be scanned out to the SDB. (See Example 22-1.)

#### Example 22-1. Integer Register File Read

```
// copy XREG to mdout; scan it out.
scan_in("sta %XREG, [%g0] mdout", mci, poll)
scan_out(mdout, SDB)
```

**22.6.3.2 Integer Register File Write**

To store a new value (or restore an old value) into an integer register at the CWP, the sequence in Example 22-2 can be used.

**Example 22-2. Integer Register File Write**

```
// Scan in new value; install it into integer register.
scan_in(new_integer_register_file_value,mdin)
scan_in("lda [%g0] mdin, %XREG", mci, poll)
```

**22.6.3.3 Integer State Register Read**

The sequence in Example 22-3 will transfer any of the integer state registers (PSR, WIM, TBR, and Y) into the MDOUT register. The sequence in Example 22-3 assumes that access to the PSR is desired. Note the use of the MTMP1 register to preserve and restore integer register state.

**Example 22-3. Integer State Register Read**

```
// prologue
scan_in("sta %g1, [%g0] mtmpl", mci, poll)
// copy PSR to mdout through g1 scan it out.
scan_in("rd %psr, %g1", mci, poll)
scan_in("sta %g1, [%g0] mdout", mci, poll)
scan_out(mdout, SDB)
// epilogue
scan_in("lda [%g0] mtmpl, %g1", mci, poll)
```

**22.6.3.4 Integer State Register Write**

The sequence in Example 22-4 will modify any integer state register. The sequence in Example 22-4 modifies the PSR as an example.

**Example 22-4. Integer State Register Write**

```
// prologue
scan_in("sta %g1, [%g0] mtmpl", mci, poll)
// Scan in new value to g1 and install into psr.
scan_in(new_integer_register_file_value,mdin)
scan_in("lda [%g0] mdin, %g1", mci, poll)
scan_in("wr %g1, %psr", mci, poll)
//epilogue
scan_in("lda [%g0] mtmpl, %g1", mci, poll)
```

**22.6.3.5 Memory Read**

The sequence in Example 22-5 demonstrates reading a signed byte value at a given memory address XADDR within an implicit alternate address space (supervisor data space where ASI=0xb). This ASI is used because the processor is effectively executing in supervisor mode.

### Example 22-5. Memory Read

```
//prologue
scan_in("sta %g1, [%g0] mtmp1", mci, poll)
scan_in("sta %g2, [%g0] mtmp2", mci, poll);
// Scan in xaddr. copy to g1
scan_in(new_xaddr_value,mdin)
scan_in("lda [%g0] mdin, %g1", mci, poll)
// load g2 w/ *xaddr; copy to mdout. Scan it out.
scan_in("ldsb [%g1], %g2", mci, poll)
scan_in("sta %g2, [%g0] mdout", mci, poll)
scan_out(mdout, SDB)
//epilogue
scan_in("lda [%g0] mtmp1, %g1", mci, poll)
scan_in("lda [%g0] mtmp2, %g2", mci, poll);
```

At the conclusion of the sequence, the value of the signed byte residing in memory address XADDR within the implicit supervisor data space ASI (0xb) will be placed in the MDOUT register. When MSTAT.MIDONE is asserted, the data can be scanned out to the SDB.

The sequences for signed and unsigned half-word and word and double-word reads are similar. Reads from alternate address spaces are also similar.

### 22.6.3.6 Write Memory

The sequence in Example 22-6 modifies memory by writing a byte value at a given memory address XADDR within an implicit alternate address space (supervisor data space where ASI=0xb).

### Example 22-6. Write Memory

```
// prologue
scan_in("sta %g1, [%g0] mtmp1", mci, poll)
scan_in("sta %g2, [%g0] mtmp2", mci, poll);
// Scan in XADDR value. copy \s-2XADDR\s0 to g1.
scan_in(XADDR_VALUE, mdin)
scan_in("lda [%g0] mdin, %g1",mci, poll)
// Scan in NEW_DATA value. copy NEW_DATA to g2.
scan_in(NEW_DATA_VALUE, mdin)
scan_in("lda [%g1] mdin, %g2",mci, poll)
// install NEW_DATA at XADDR.
scan_in("stb %g2, [%g1]",mci, poll)// epilogue
scan_in("lda [%g0] mtmp1, %g1", mci, poll)
scan_in("lda [%g0] mtmp2, %g2", mci, poll)
```

During a memory write, MDIN will be used twice by the SDB. Initially the SDB will scan into MDIN the value of the memory address XADDR to be written. Once this address is transferred to the integer register file, the SDB will then scan in the new value (NEW\_DATA\_VALUE) to be written at the specified memory location (XADDR).

The sequences for half-word, word, and double-word writes are very similar. Writes to alternate spaces are also similar, with the STB instruction replaced by an STBA.

### 22.6.3.7 Floating-Point Register Read

For simplicity, Example 22-7 provides the same address in register MDIN; its selection is assumed to be safe. The sequence in Example 22-7 will transfer a floating-point register XREG into the MDOUT register for the SDB to scan out. The sequence for reading floating-point control registers is similar.

#### Example 22-7. Floating-Point Register Read

```
// prologue
scan_in("sta %g1, [%g0] mtmp1", mci, poll)
scan_in("sta %g2, [%g0] mtmp2", mci, poll)
// Scan address of background memory word (XADDR).
// copy XADDR into %g1; load *xaddr into %g2.
scan_in(xaddr_value, mdin);
scan_in("lda [%g0] mdin, %g1", mci, poll)
scan_in("ld [%g1], %g2", mci, poll)
// this scan-based debug primitive MUST not change non-
// scan-based debug
// memory state. FP reads require we alter and then
// restore a background memory state. No sufficient number
// of mtmp to do this so we augment the storage by
// scanning out to SDB memory.
scan_in("sta %g2, mdout", mci, poll)
scan_out(mdout, SDB_temp1)
// before issuing next SDB instruction,
// make sure mstat.fqe=1 and mstat.pfpx=0
// copy fp entry to bgnd word; copy into iu temp.
scan_in("st %fXREG, [%g1]", mci, poll)
scan_in("ld [%g1], %g2", mci, poll)
// copy FP entry to mdout; scan out fp entry.
scan_in("sta %g2, [%g0] mdout", mci, poll)
scan_out(mdout, SDB)
// restore original background memory word
scan_in(SDB_temp1, mdin)
scan_in("lda [%g0] mdin, %g2", mci, poll)
scan_in("st %g2, [%g1]", mci, poll)
// epilogue
scan_in("lda [%g0] mtmp1, %g1", mci, poll)
scan_in("lda [%g0] mtmp2, %g2", mci, poll)
```

### 22.6.3.8 Floating-Point Register Write

The sequence for writing floating-point registers is similar to the reading sequence, except that a floating-point register is loaded from memory rather than written. Example 22-8 is a possible code sequence.

**Example 22–8. Floating-Point Register Write**

```

// prologue
scan_in("sta %g1, [%g0] mtmp1", mci, poll)
scan_in("sta %g2, [%g0] mtmp2", mci, poll)

// Scan address of background memory word (XADDR).
// copy XADDR into %g1; load *xaddr into %g2.
scan_in(XADDR_VALUE, mdin)
scan_in("lda [%g0] mdin, %g1", mci, poll)

scan_in("ld [%g1], %g2", mci, poll)

// preserve original background word in
// SDB_templ. This scan-based debug primitive
// must not change non-scan-based debug memory state. FP
// writes require we alter and then restore
// a background memory state. No sufficient number
// of mtmp to do this so we augment the storage by
// scanning out to SDB memory.
scan_in("sta %g2, mdout", mci, poll)
scan_out(mdout, SDB_templ)

// Scan in new new fp_data_value
// load it into iu rfile. write it to background word.
scan_in(NEW_FP_DATA_VALUE, mdin)
scan_in("lda [%g0] mdin, %g2", mci, poll)
scan_in("st %g2, [%g1]", mci, poll)

// before issuing next scan-based debug instruction,
// make sure mstat.fqe=1 and mstat.pfpx=0
// install new value into FP register.
scan_in("ld [%g1], %fXREG", mci, poll)

// restore original background memory word.
scan_in(SDB_templ, mdin)
scan_in("lda [%g0] mdin, %g2", mci, poll)
scan_in("st %g2, [%g1]", mci, poll)

// epilogue
scan_in("lda [%g0] mtmp1, %g1", mci, poll)
scan_in("lda [%g0] mtmp2, %g2", mci, poll)

```

**22.6.3.9 Floating-Point State Register Read**

Reading values from floating-point-state registers, such as the FSR, is similar to the previous two examples. The floating-point memory references are replaced by store FSR operations. Extracting entries from the floating-point queue is more difficult. While in scan-based debug mode, FQ entries should never be extracted unless MSTAT.PFPX indicates an FP exception. Otherwise, the remaining FPOPs in the FQ should be allowed to execute. If PFPX is asserted and the SDB wants to recover, the FQ can be read with a double-word size. Since MDOUT is only a single-word wide, two scan passes are required to transfer the full queue entry to the SDB.

There is an additional architectural side effect of reading the floating-point queue. When an entry is read, it is effectively removed from the queue. Since scan-based debug is required to be non-intrusive to program execution, the old state of the queue must be restored. If the queue state is to remain unchanged after being observed, all entries in the queue must be extracted using STDFQ until it is empty. All these removed entries must then be reinserted (re-written) back into the queue. Although there is no simple instruction for writing to the FP queue, it may be restored by executing a floating-point instruction while in scan-based debug mode. The instruction and PC value are both stored in the queue. By writing the PC into the MDIN register and then issuing the floating-point instruction as a scan-based debug instruction, the queue will be restored.

#### **22.6.3.10 Floating-Point State Register Write**

Writing to the floating-point state registers is similar to reading them. The same restrictions apply.

#### **22.6.3.11 Setting Code and Data Address Breakpoints**

The following sequence will set up a code or data address breakpoint and resume normal execution. When the breakpoint occurs, the processor will re-enter scan-based debug mode.

A string of scan-based debug instruction sequences will be required to write the code address breakpoint register set.

- 1) Write the desired code or data space breakpoint address value (virtual or physical) into the Breakpoint Address Register (BKV).
- 2) Write the code or data space breakpoint address compare mask into the Breakpoint Mask Register (BKM).
- 3) Write the breakpoint control register (BKC) to clear BKC.CBKEN and select the desired values for BKC.CSPACE, BKC.PAMD, BKC.CBKEN, BKC.DBREN, and BKC.DBWEN.
- 4) Write the breakpoint status register (BKS) to clear any prior code breakpoint status.
- 5) Write the action on event control register (ACTION) so that the desired breakpoint event will generate a scan-based debug mode request and (optionally) assert a scan-based debug strobe (ESB) pin. See Subsection 15.2.5 for more details.

Example 22-9 illustrates such a scan-based debug mode request.



### Example 22-9. Scan-Based Debug Mode Request

```
// prologue
scan_in("sta %g1, [%g0] mtmp1", mci, poll)
scan_in("sta %g2, [%g0] mtmp2", mci, poll)
scan_in("sta %g3, [%g0] mdout", mci, poll)
scan_out(mdout, SDB_temp1)

// set g1 to ASI cbkv addr offset w/in MDIAG ASI;
scan_in("or %g0, bkv, %g1", mci, poll)

// Scan-in lower 32-bit (of 36 bits) for next cbkv value
// install it in g2 of register pair g[23].
scan_in(NEW_CBKV_VALUE_LO32, mdin)
scan_in("lda [%g0] mdin, %g2", mci, poll)

// Scan-in upper 4-bit portion (of 36-bits)
// for next cbkv value
// install it in g3 of register pair g[23].
scan_in(NEW_CBKV_VALUE_HI4, mdin)
scan_in("lda [%g0] mdin, %g3", mci, poll)

// install register pair g[23] into mdiag cbkv.
scan_in("stda %g2, [%g1] mdiag", mci, poll)

// set g1 to ASI cbkm addr offset w/in MDIAG ASI;
scan_in("or %g0, bkm, %g1", mci, poll)

// Scan-in lower 32-bit portion (of 36-bits)
// for next cbkm value
// install it in g2 of register pair g[23].
scan_in(NEW_CBKM_VALUE_LO32, mdin)
scan_in("lda [%g0] mdin, %g2", mci, poll)

// Scan-in upper 4-bit portion (of 36-bits)
// for next cbkm value
// install it in g3 of register pair g[23].
scan_in(NEW_CBKM_VALUE_HI4, mdin)
scan_in("lda [%g0] mdin, %g3", mci, poll)

// install g[23] to cbkm
scan_in("stda %g2, [%g1] mdiag", mci, poll)

// set g1 to addr cbkc;
scan_in("or %g0, bkc, %g1", mci, poll)

// Scan-in NEW_CBKC_VALUE.
// read it into the integer_register_file; install it into
// CBKC.
scan_in(NEW_CBKC_VALUE, mdin)
scan_in("lda [%g0] mdin, %g2", mci, poll)
scan_in("sta %g2, [%g1] mdiag", mci, poll)

// set g1 to addr cbks in mdiag; clear cbks.
scan_in("or %g0, bks, %g1", mci, poll)
scan_in("sta %g0, [%g1] mdiag", mci, poll)
```

**Example 22–9. Scan-Based Debug Mode Request (Continued)**

```
// scan-in NEW_ACTION_VALUE.
scan_in(NEW_ACTION_VALUE, mdin);
// read it into the integer register file; install it into
// ACTION.
scan_in("lda [%g0] mdin, %g2", mci, poll)
scan_in("sta %g2, [%g0] action", mci, poll)
// epilogue
scan_in("lda [%g0] mtmp1, %g1", mci, poll)
scan_in("lda [%g0] mtmp2, %g2", mci, poll)
scan_in(SDB_templ, mdin)
scan_in("lda [%g0] mdin, %g3", mci, poll)
```

Setting a breakpoint on a specific data memory address reference is very similar to the above sequence. The only difference is that the value is written into memory-mapped BKC and ACTION registers. In the above example, BKC.CSPACE, BKC.CBKEN, and ACTION.I\_CBK are set. A write-only data address breakpoint would clear BKC.CSPACE, BKC.DBFEN, BKC.DBREN, and ACTION.I\_DBK, while BKC.DBWEN would be set.

**22.6.3.12 "Run-for-N" Instructions/Cycles**

Sequences for programming "Run-for-N" instructions and cycles are similar to setting code/data breakpoints. (See Example 22–10.) The cycle counter breakpoint is most useful for statistically profiling execution of a program. The instruction counter breakpoint is useful for single-stepping or block-stepping through a program execution.

**Example 22-10. "Run-for-N" Instruction Cycles**

```
// prologue
scan_in("sta %g1, [%g0] mtmp1", mci, poll)
scan_in("sta %g2, [%g0] mtmp2", mci, poll)
scan_in("sta %g3, [%g0] mdout", mci, poll)
scan_out(mdout, SDB_templ)

// set g1 to ASI address for CNTV w/in EDIAG.
scan_in("or %g0, 0x000, %g1", mci, poll)

// scan-in NEW_CNTV_VALUE for ICNT/CCNT.
// read it into %g2; install into CNTV.
scan_in(NEW_CNTV_VALUE, mdin);
scan_in("lda [%g0] mdin, %g2", mci, poll)
scan_in("sta %g2, [%g1] cntv", mci, poll)

// clear cnts
scan_in("sta %g0, [%g0] cnts", mci, poll)

// scan-in NEW_ACTION_VALUE for ACTION.
// read it into %g2; install it into ACTION.
scan_in(NEW_ACTION_VALUE, mdin);
scan_in("lda [%g0] mdin, %g2", mci, poll)
scan_in("sta %g2, [%g1] action", mci, poll)

// scan-in NEW_CNTRC_VALUE for ICNTEN/CCNTEN.
// read it into %g2; install it into CNTRC.
scan_in(NEW_CNTRC_VALUE, mdin)
scan_in("lda [%g0] mdin, %g2", mci, poll)
scan_in("sta %g2, [%g1] cntc", mci, poll)

// epilogue
scan_in("lda [%g0] mtmp1, %g1", mci, poll)
scan_in("lda [%g0] mtmp2, %g2", mci, poll)
scan_in(SDB_templ, mdin)
scan_in("lda [%g0] mdin, %g3", mci, poll)
```

## **22.7 Approximate Latencies for Each SDB Primitive**

Each scan-based debug instruction execution requires several multi-bit JTAG TDR scan operations.

The number of scan-based debug instructions required per primitive is:

- ☐ One to access an integer register entry.
- ☐ Four to access an integer control register.
- ☐ Seven to access a memory location.
- ☐ 13 to access a floating-point register.
- ☐ 13 to set up an instruction (cycle) counter expiration.
- ☐ 21 to set up an instruction (data) address breakpoint.

The number of MDIN (or MDOUT) scan operations required per SDB primitive is:

- ☐ One scan operation per integer register access.
- ☐ One MDIN scan operation per floating-point register write.
- ☐ Two (MDIN and MDOUT) scan operations per memory access.
- ☐ Three MDIN scan operations per setting of a breakpoint or counter event.

Each scan-based debug instruction scan in and scan-based debug status scan out requires about 50 TCK cycles. Each MDIN scan in takes about 40 TCK cycles. Each MDOUT scan out takes about 40 TCK cycles.



# Clocking

---

Proper clocking is essential at high operating frequencies. This chapter will describe essential clock requirements for the SuperSPARC processor (SSP) and MultiCache Controller (MXCC).

In order to reduce system clock skew, a phase locked loop (PLL) is employed for each of the clock inputs.

Topic	Page
23.1 Phase Locked Loop Operation .....	23-2
23.2 Input Clock Requirements .....	23-6
23.3 MXCC Synchronous and Asynchronous Operation .....	23-7

## 23.1 Phase Locked Loop Operation

The SuperSPARC chips use PLLs to reduce the skew between the clock inputs and points internal to the chips where the clock signals are used. The reduced internal skews allow the timing specifications to be tighter than they would be without the use of the PLLs. Tighter timing specifications simplify the design and construction of high-speed systems.

Each PLL operates by constantly measuring internal clock routing delay and internally generating a clock that is effectively ahead of the external clock by an amount equal to the internal routing delay. This ensures that internal logic sees a clock signal with very low skew from the external clock pin.

Figure 23-1 shows the scheme used for reducing skew. The PLL samples the input clock and the clock at the end of a balanced distribution tree. The phase comparator in the PLL adjusts the phase of the voltage-controlled oscillator (VCO) so that the two clocks have the same frequency and phase.

---

**Note:**

Prior to normal operation, the PLL must be allowed time to stabilize (i.e., after power-up or when PLL has been disabled). During this time, RESET should be active. The time required is 100 milliseconds.

---

---

**Note:**

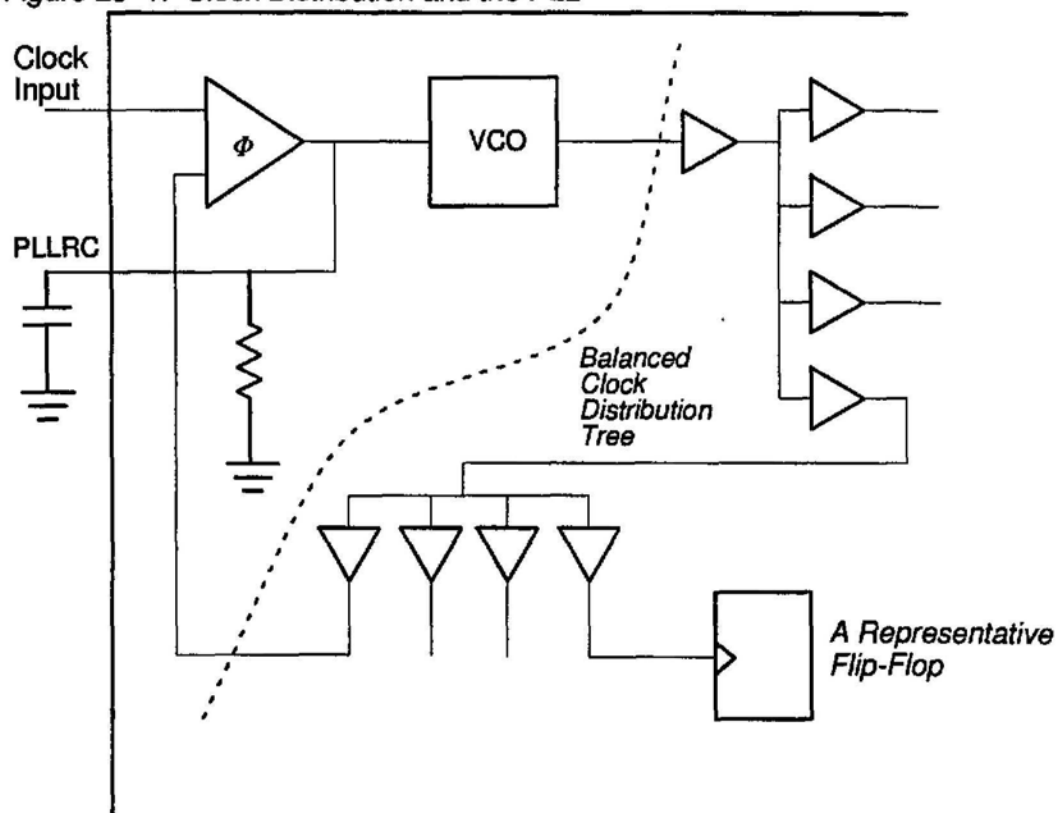
The JTAG TAP controller must be reset prior to or at the same time as RESET in order for the PLL to begin initialization. The TAP controller may be initialized either by asserting the TRST pin or by asserting the TMS pin for five consecutive cycles of test clock (TCK). If this reset does not occur, the PLL clock feedback loop may not be established, and unpredictable operation may result.

---

Whenever the JTAG interface is not in use by a particular system, asserting the TRST signal statically is strongly recommended.

The input clock should never be stopped or changed from its normal periodic operation while the PLL is enabled. Doing so will cause PLL instability and unpredictable operation. If the clock is changed from its normal regular pattern, the change must occur only while RESET is asserted. RESET must remain asserted for the stabilization period of at least 100 ms after the clock resumes regular periodic operation, regardless of whether the frequency is changed.

Figure 23-1. Clock Distribution and the PLL



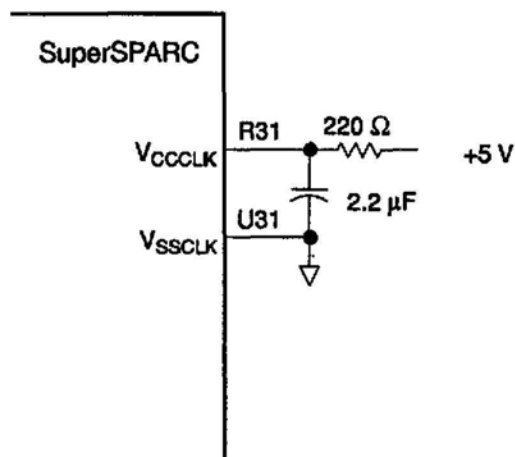
### 23.1.1 SuperSPARC

To guarantee PLL stabilization, **RESET** should be active for at least 100ms after power and clock become stable.

The operation of the PLL circuit can be disrupted by noise in its power supply. To ensure proper operation of the PLL clock, system noise should be filtered out of  $V_{CCCLK}$  and  $V_{SSCLK}$ . Figure 23-2 shows a recommended filter circuit.



Figure 23–2. Typical Phase Locked Loop (PLL) Filter Circuit for SuperSPARC Processor



### 23.1.2 MXCC

PLL relies on an external loop filter capacitor to integrate phase comparisons. An external capacitor must be connected between PLLRC and ground. The recommended value is 0.1 μF.

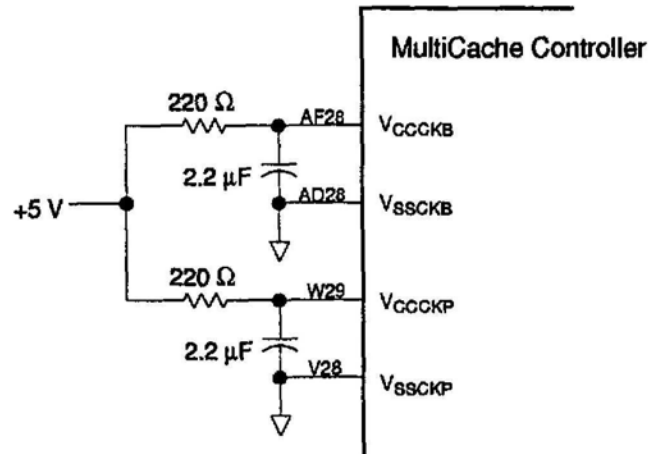
The MXCC has two clock inputs, BCLK and PCLK. It therefore has two PLLs. Only the PLL bypass control pin (PLLBYP) is shared by the two clocks.

To guarantee PLL stabilization, RSTIN should be asserted for at least 100 ms after the power and clocks are stable to the MXCC.

The operation of the PLL circuits can be disrupted by noise on the power supply.

To ensure proper operation of the PLL clock, system noise should be filtered from VCCCKB, VCCCKP, VSSCKB and VSSCKP. Figure 23–3 shows a recommended filter circuit.

Figure 23–3. Typical Phase Locked Loop (PLL) Filter Circuit for MXCC



The PLLs rely on external loop filter capacitors to integrate phase comparisons. External capacitors must be connected between PPLLRC and ground and between BPLLRC and ground. The recommended value for each of the capacitors is 0.1μF.

## **23.2 Input Clock Requirements**

The SSP and the MXCC can tolerate most clean stable clock sources when the PLL is enabled. With the PLL enabled, the chips use only the rising edge of the input clocks. Internally, the processors perform as indicated below.

### **23.2.1 SuperSPARC**

The SSP multiplies, then divides the clock to provide a stable 50% duty cycle clock. Input duty cycle must be at least 25% (either high or low). When the PLL is bypassed, care must be taken to provide a 50% duty cycle clock. Pin timings for operation with the PLL bypassed are not defined.

### **23.2.2 MultiCache Controller**

The MXCC doubles the frequency of the input clocks and then halves them to produce stable clocks with 50% duty cycles. The high time of the input clocks must be between 25% and 75%.

### **23.3 MXCC Synchronous and Asynchronous Operation**

The MXCC has two clock inputs, PCLK and BCLK. PCLK provides timing to circuits on the processor side of the chip, including the VBus interface, E-cache control, and E-cache tags. BCLK provides timing to circuits on the system bus side of the chip, including the MBus and XBus interfaces. Data traversing between the two clock domains passes by way of FIFO queues. Control signals traversing between domains pass through synchronizers.

This organization allows the processor clock to be faster than the bus clock and allows for modular processor upgrades without affecting the rest of the system.

MXCC allows for either synchronous or asynchronous operation controlled by the  $\overline{\text{SYNC}}$  pin. The  $\overline{\text{SYNC}}$  pin should not be changed except while  $\overline{\text{RSTIN}}$  is asserted.

#### **23.3.1 Asynchronous Operation**

When PCLK is faster than BCLK, MXCC must be operated asynchronously. Asynchronous operation is selected by deasserting the  $\overline{\text{SYNC}}$  pin (H). Due to the design of the internal synchronizers, PCLK must be at least 10% faster than BCLK, and the ratio of PCLK to BCLK must not exceed 5 to 1.

#### **23.3.2 Synchronous Operation**

Synchronous operation is selected when the  $\overline{\text{SYNC}}$  pin is asserted (L). In synchronous operation, the synchronizers on control signals between the two clock domains are defeated. For proper operation, BCLK and PCLK must be connected to the same clock with a very slow skew between them (a maximum of 150ps of skew between them is recommended).



## **MBus Module**

---

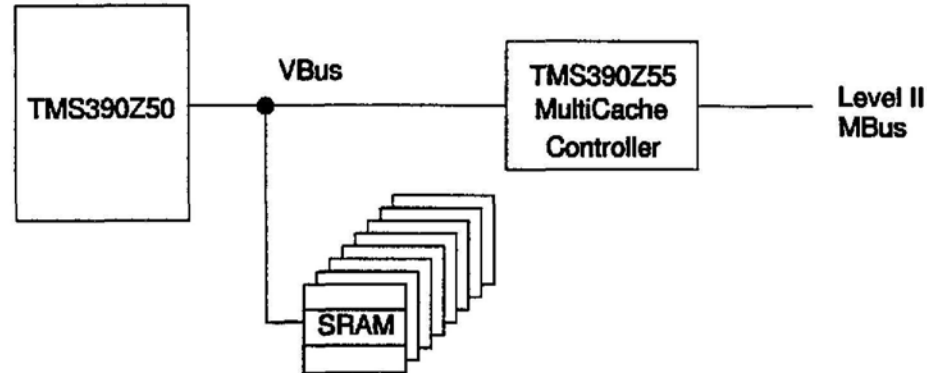
This chapter explains how the MBus conversion module connects a Super-SPARC processor (SSP) and MultiCache Controller (MXCC) together for use on the MBus.

<b>Topic</b>	<b>Page</b>
<b>24.1 Full Module MBus System</b> .....	<b>24-2</b>
<b>24.2 MBus Module Schematics</b> .....	<b>24-3</b>
<b>24.3 Mechanical Information</b> .....	<b>24-9</b>

## 24.1 Full Module MBus System

The Full Module MBus system is diagrammed in Figure 24-1. The external cache memory provides significant performance improvement and greatly decreases bus traffic in order to support more processors on a system bus.

Figure 24-1. Full MBus Module Diagram



The E-cache is organized as a direct-mapped cache with a normal size of 1M-byte. This configuration is implemented with eight 128Kx8 or 128Kx9 synchronous SRAMs. To implement byte parity on the E-cache data storage the 128K x 9 SRAMs are needed. Parity is directly supported by both the TMS390Z50 and TMS390Z55.

Synchronous SRAMs have registers on each input and output. This allows pipelined operation. An address is presented to an SRAM before the active clock edge, and it is registered in the SRAM at the clock edge. The SRAM reads out the addressed location before the next active clock edge, and the result is stored in an output register at the edge. New addresses can be supplied at each clock edge, and new outputs appear after one clock period of delay. Writing works similarly with address, data, output enable, and write-enable being registered on the active clock edge and stored into the internal array during the subsequent clock period.

The synchronous SRAMs used in the Full Module configurations are available from several manufacturers.

## **24.2 MBus Module Schematics**

The schematics of the MBus Module are presented in Figure 24-2 through Figure 24-6.



Figure 24-2. Full Module Schematic Diagram (sheet 1 of 5)

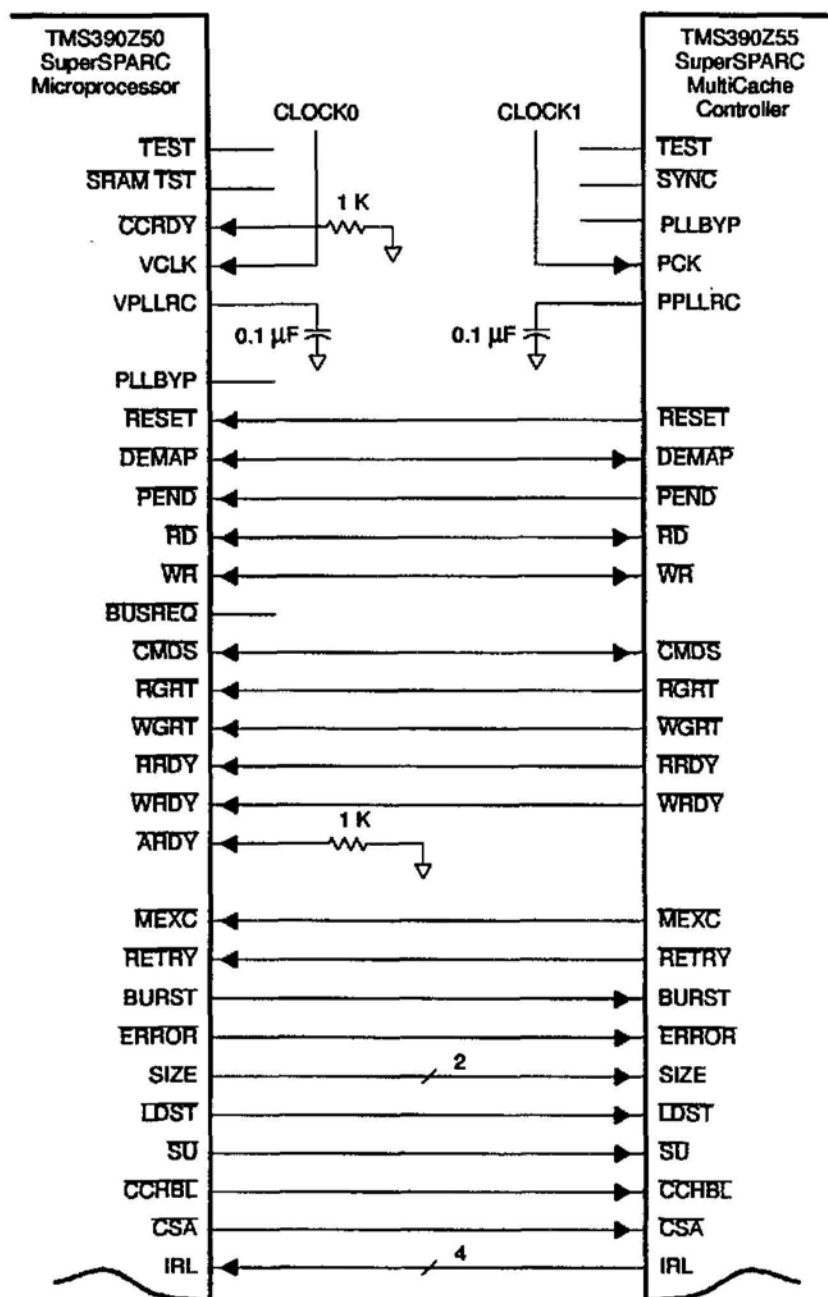


Figure 24-3. Full Module Schematic Diagram (sheet 2 of 5)

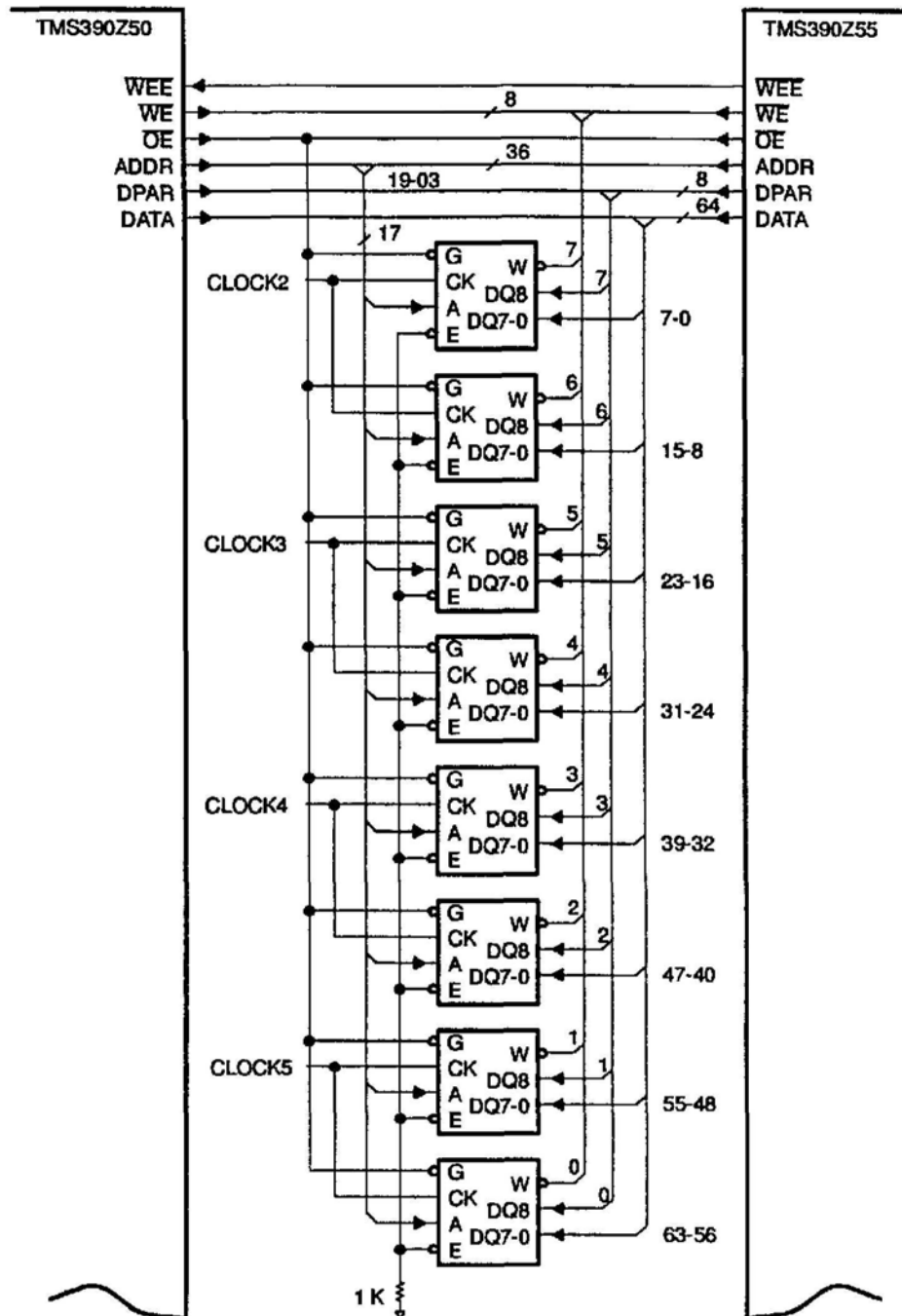


Figure 24-4. Full Module Schematic Diagram (sheet 3 of 5)

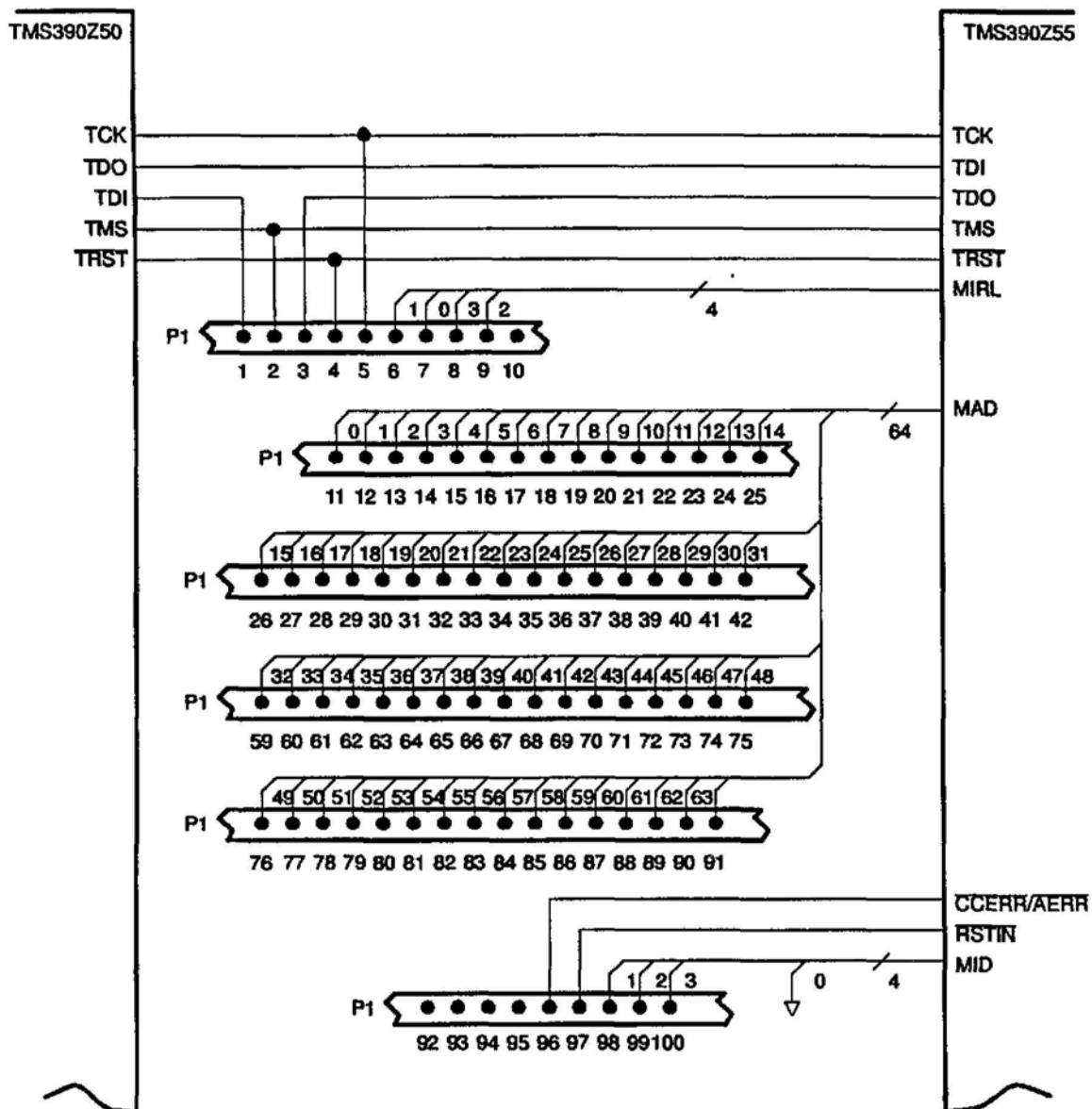


Figure 24-5. Full Module Schematic Diagram (sheet 4 of 5)

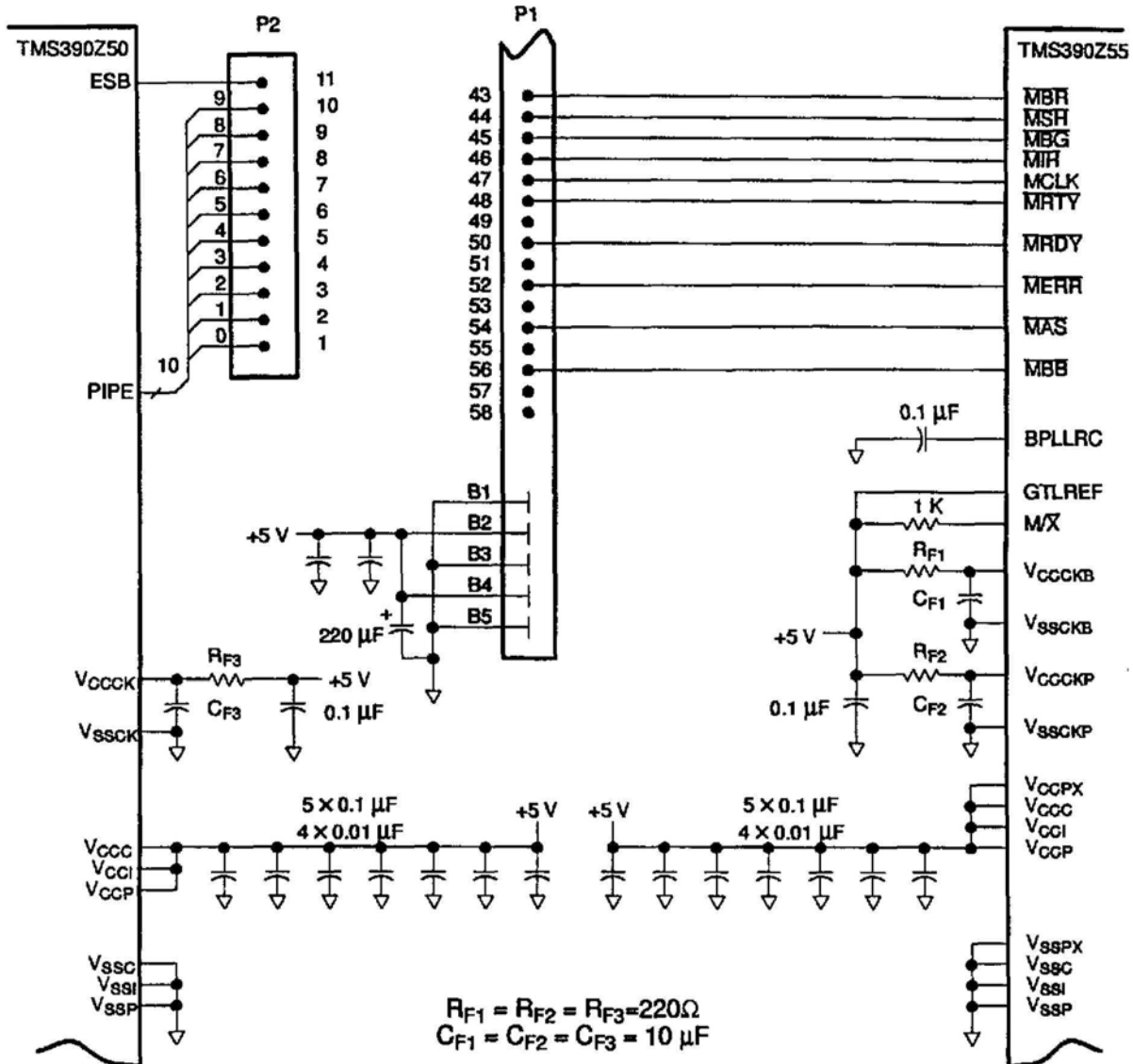
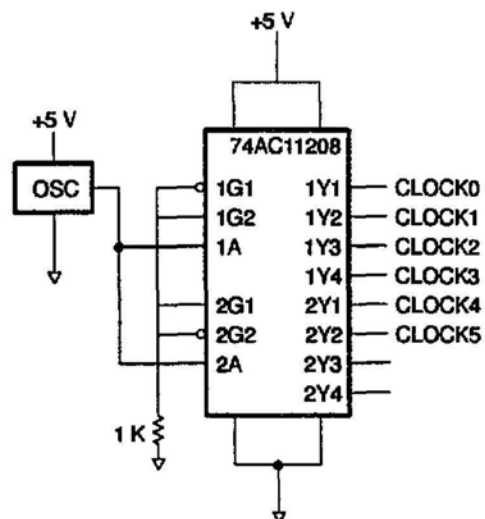


Figure 24-6. Full Module Schematic Diagram (sheet 5 of 5)



### 24.3 Mechanical Information

See Section 7 of *SPARC MBus Interface Specification*.

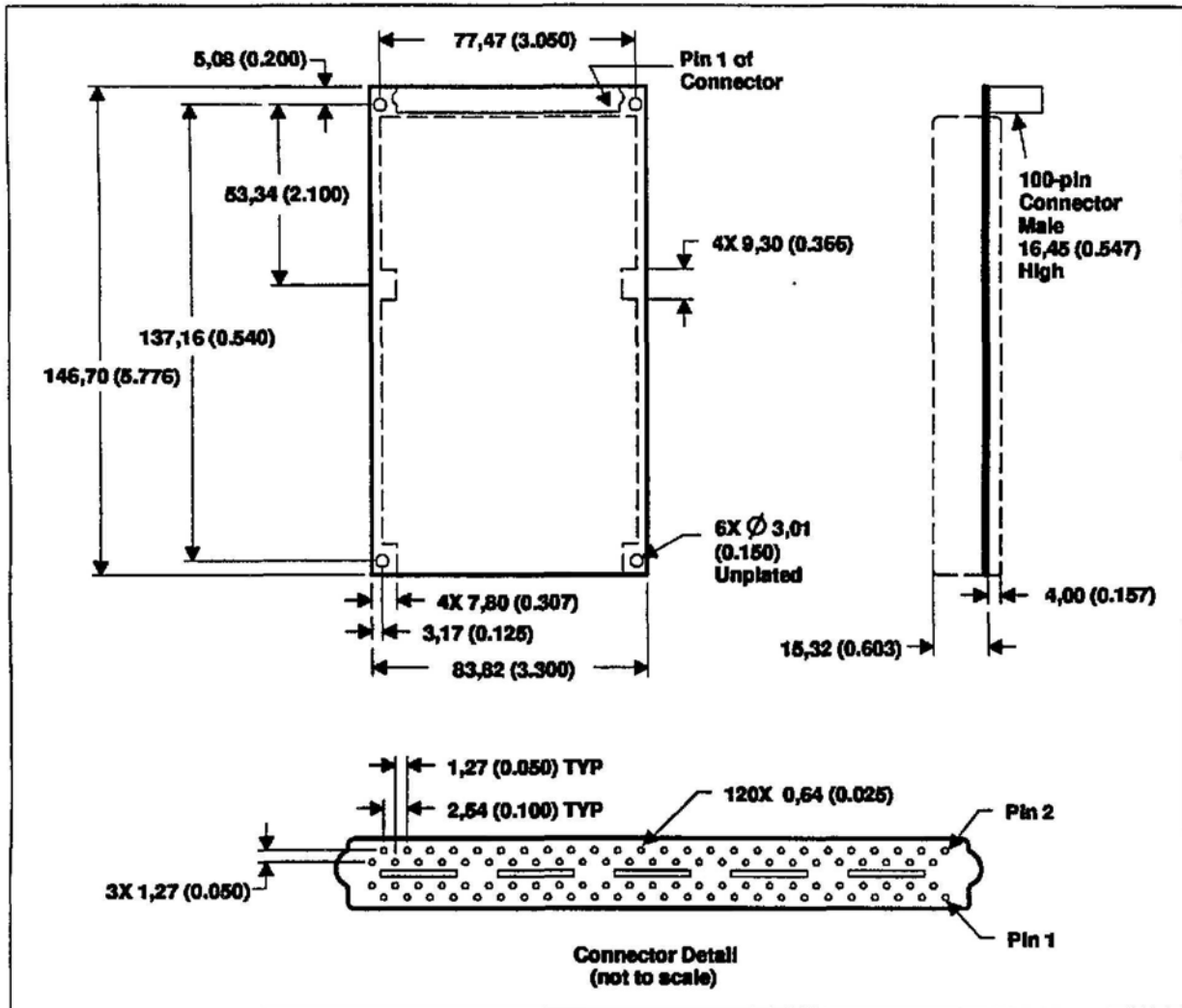
The dimensions of modules are summarized in and shown in Table 24-1. Through-hole components, components with heatsinks, heatsinks, and other tall components that may be on a module will be mounted on the top side of the module, opposite from the connector. Only low-profile, surface-mount components, if any, will be mounted on the bottom (connector) surface.

Table 24-1. Dimensions of MBus Modules

Dimension	Full-Size Module	Units
Width	83.82	mm
	3.30	in
Length	146.70	mm
	5.776	in
Clearance Above	15.32	mm
	0.603	in
Clearance Below	4.00	mm
	0.157	in

The module mechanical envelope is shown in Figure 24-7.

Figure 24-7. Module Mechanical Envelope



The connector types are given in Table 24-2. The T1 module contains a plug, and the system board should contain a mating receptacle. T1 modules may use the listed connectors or other equivalent connectors.

Table 24-2. Module Connectors

Connector Type	Maker	Part Number
Plug	Fujitsu	FCN 264P100-G/O
Plug	AMP	1121354-4
Receptacle for 0.062 PCB	Fujitsu	FCN264J100-G/O

## **Instruction Summary**

---

This appendix contains a summary of the SuperSPARC processor's (SSP's) instruction set. The instructions are formatted to allow easy decoding and encoding.

<b>Topic</b>	<b>Page</b>
<b>A.1 Instruction Fields .....</b>	<b>1-2</b>
<b>A.2 Format 1 .....</b>	<b>1-4</b>
<b>A.3 Format 2 .....</b>	<b>1-5</b>
<b>A.4 Format 3 .....</b>	<b>1-7</b>



## A.1 Instruction Fields

<b>op</b>	This two-bit field encodes all three major formats.
<b>op2</b>	This three-bit field encodes format 2 instructions.
<b>op3</b>	This six-bit field encodes format 3 instructions.
<b>opf</b>	This nine-bit field encodes format 3 floating-point operate (FPop) instructions.
<b>a</b>	A value of 1 in this one-bit field annuls the execution of the instruction that follows a conditional or an unconditional taken branch.
<b>cond</b>	This four-bit field selects the condition code(s) to test for a branch or trap instruction.
<b>rd</b>	This five-bit field is the address of a destination (or source) <i>r</i> or <i>f</i> register(s) used in a load (or store) or arithmetic instruction. For instructions that read or write a double (or quad), the least significant one or two bits are unused and should be zero.
<b>rs1</b>	This five-bit field is the address of the first <i>r</i> or <i>f</i> register(s) source operand. For instructions that read a double (or quad), the least significant one (or two) bits are unused and should be zero.
<b>rs2</b>	This five-bit field is the address of the second <i>r</i> or <i>f</i> register(s) source operand when the operand is not an immediate operation. For instructions that read a double (or quad), the least significant one or two bits are unused and should be zero.
<b>slmm13</b>	This 13-bit field is a sign-extended 13-bit immediate value used as the second ALU operand for a load or store instruction or for an integer arithmetic instruction.
<b>imm22</b>	This 22-bit field is a constant that the SETHI instruction places in the upper end bits of a destination register.
<b>disp30</b>	This 30-bit field represents a word-aligned, sign-extended, PC-relative displacement for a call instruction.
<b>asi</b>	This eight-bit field is the address space identifier supplied to a load or a store alternate instruction.

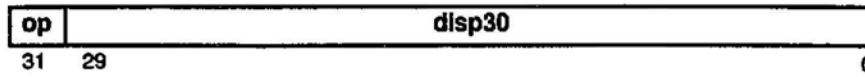
**Note:**

- † Denotes a privileged instruction.
- ‡ Denotes a privileged instruction if the ASR register referenced in the instruction is privileged.
- The SuperSPARC processor (SSP) does not support quad-precision operands. These instructions will cause an *unimplemented\_FPop* trap; however, they can be implemented in software.
- ★ Denotes an instruction specific to the SSP.

*Format 1*

---

**A.2 Format 1 (op = 01)**



01 ..... **CALL**

**A.3 Format 2 (op = 00)**

op	reserved	op2	const22
31	29	24	21
			0

00 00000 000 ..... UNIMP

op	a	cond	op2	disp22
31	29	28	24	21
				0

00	A	0000	010	.....	BN{,A}
00	A	0001	010	.....	BE{,A}
00	A	0010	010	.....	BLE{,A}
00	A	0011	010	.....	BL{,A}
00	A	0100	010	.....	BLEU{,A}
00	A	0101	010	.....	BCS{,A}
00	A	0110	010	.....	BNEG{,A}
00	A	0111	010	.....	BVS{,A}
00	A	1000	010	.....	BA{,A}
00	A	1001	010	.....	BNE{,A}
00	A	1010	010	.....	BG{,A}
00	A	1011	010	.....	BGE{,A}
00	A	1100	010	.....	BGU{,A}
00	A	1101	010	.....	BCC{,A}
00	A	1110	010	.....	BPOS{,A}
00	A	1111	010	.....	BVC{,A}

op	rd	op2	imm22
31	29	24	21
			0

00 00000 100 00000000000000000000 ..... NOP  
 00 100 ..... SETHI

# Format 2

op	a	cond	op2	disp22
31	29	28	24	21
				0

00	A	0000	110	.....	FBN{,A}
00	A	0001	110	.....	FBNE{,A}
00	A	0010	110	.....	FBLG{,A}
00	A	0011	110	.....	FBUL{,A}
00	A	0100	110	.....	FBL{,A}
00	A	0101	110	.....	FBUG{,A}
00	A	0110	110	.....	FBG{,A}
00	A	0111	110	.....	FBU{,A}
00	A	1000	110	.....	FBA{,A}
00	A	1001	110	.....	FBE{,A}
00	A	1010	110	.....	FBUE{,A}
00	A	1011	110	.....	FBGE{,A}
00	A	1100	110	.....	FBUGE{,A}
00	A	1101	110	.....	FBLE{,A}
00	A	1110	110	.....	FBULE{,A}
00	A	1111	110	.....	FBO{,A}

**A.4 Format 3 (op = 10 or op = 11)**

op	rd	op3	rs1	1	simm13	
op	rd	op3	rs1	0	00000000	rs2
31	29	24	18	13	12	4 0

10	000000	ADD
10	000001	AND
10	000010	OR
10	000011	XOR
10	000100	SUB
10	000101	ANDN
10	000110	ORN
10	000111	XORN
10	001000	ADDX
10	001010	UMUL
10	001011	SMUL
10	001100	SUBX
10	001110	UDIV
10	001111	SDIV
10	010000	ADDCC
10	010001	ANDCC
10	010010	ORCC
10	010011	XORCC
10	010100	SUBCC
10	010101	ANDNCC
10	010110	ORNCC
10	010111	XORNCC
10	011000	ADDXCC
10	011010	UMULCC
10	011011	SMULCC
10	011100	SUBXCC
10	011110	UDIVCC
10	011111	SDIVCC
10	100000	TADDCC
10	100001	TSUBCC
10	100010	TADDCCCTV
10	100011	TSUBCCCTV
10	100100	MULSCC
10	100101	SLL
10	100110	SRL
10	100111	SRA

### Format 3

op	rd	op3	rs1	0	00000000000000		
31	29	24	18	13	12	4	0

10		101000	00000	.....	RDY
10		101000	rs1 ≠ 0	.....	RDASR†
10	00000	101000	01111	.....	STBAR
10	00000	101000	11111	.....	SIGM★
10		101001	.....	.....	RDPSR†
10		101010	.....	.....	RDWIM†
10		101011	.....	.....	RDTBR†

op	rd	op3	rs1	1	simm13		
op	rd	op3	rs1	0	00000000		rs2
31	29	24	18	13	12	4	0

10	00000	110000	.....	WRY
10	rd ≠ 0	110000	.....	WRASR‡
10		110001	.....	WRPSR†
10		110010	.....	WRWIM†
10		110011	.....	WRTBR†

† – privileged instruction.

‡ – privileged instruction if source/destination register is privileged.

★ – instruction specific to the SSP.

● – SuperSPARC does not support quad-precision operands. These instructions will cause an *unimplemented\_FPop* trap. These instructions may be implemented in software.

op	rd	op3	rs1	opf	rs2
31	29	24	18	13	4 0
10		110100		000000001 .....	FMOV <sub>s</sub>
10		110100		000000101 .....	FNEG <sub>s</sub>
10		110100		000001001 .....	FABS <sub>s</sub>
10		110100		000101001 .....	FSQRT <sub>s</sub>
10		110100		000101010 .....	FSQRT <sub>d</sub>
10		110100		000101011 .....	FSQRT <sub>q</sub> ●
10		110100		001000001 .....	FADD <sub>s</sub>
10		110100		001000010 .....	FADD <sub>d</sub>
10		110100		001000011 .....	FADD <sub>q</sub> ●
10		110100		001000101 .....	FSUB <sub>s</sub>
10		110100		001000110 .....	FSUB <sub>d</sub>
10		110100		001000111 .....	FSUB <sub>q</sub> ●
10		110100		001001001 .....	FMUL <sub>s</sub>
10		110100		001001010 .....	FMUL <sub>d</sub>
10		110100		001001011 .....	FMUL <sub>q</sub> ●
10		110100		001001101 .....	FDIV <sub>s</sub>
10		110100		001001110 .....	FDIV <sub>d</sub>
10		110100		001001111 .....	FDIV <sub>q</sub> ●
10		110100		001101001 .....	FsMUL <sub>d</sub>
10		110100		001101110 .....	FdMUL <sub>q</sub> ●
10		110100		011000100 .....	FITOs
10		110100		011000110 .....	FdTOs
10		110100		011000111 .....	FqTOs●
10		110100		011001000 .....	FITOd
10		110100		011001001 .....	FsTOd
10		110100		011001011 .....	FqTOd●
10		110100		011001100 .....	FITOq ●
10		110100		011001101 .....	FsTOq ●
10		110100		011001110 .....	FdTOq●
10		110100		011010001 .....	FsTOI
10		110100		011010010 .....	FdTOI
10		110100		011010011 .....	FqTOI ●



### Format 3

op	00000	op3	rs1	opf	rs2
31	29	24	18	13	4 0

10		110101		001010001 .....	FCMPs
10		110101		001010010 .....	FCMPd
10		110101		001010011 .....	FCMPq●
10		110101		001010101 .....	FCMPEs
10		110101		001010110 .....	FCMPed
10		110101		001010111 .....	FCMPEq●

† - privileged instruction.

‡ - privileged instruction if source/destination register is privileged.

★ - instruction specific to the SSP.

● - SuperSPARC does not support quad-precision operands. These instructions will cause an *unimplemented\_FPop* trap. These instructions may be implemented in software.

op	rd	op3	rs1	1	simm13					
op	rd	op3	rs1	0	00000000				rs2	
31	29	24	18	13	12			4		0

10                    111000 ..... JMPL  
 10    00000        111001 ..... RETT†

op	0	cond	op3	rs1	1	simm13				
op	0	cond	op3	rs1	0	00000000				rs2
31	29	28	24	18	13	12			4	0

10    0000    111010 ..... TN  
 10    0001    111010 ..... TE  
 10    0010    111010 ..... TLE  
 10    0011    111010 ..... TL  
 10    0100    111010 ..... TLEU  
 10    0101    111010 ..... TCS  
 10    0110    111010 ..... TNEG  
 10    0111    111010 ..... TVS  
 10    1000    111010 ..... TA  
 10    1001    111010 ..... TNE  
 10    1010    111010 ..... TG  
 10    1011    111010 ..... TGE  
 10    1100    111010 ..... TGU  
 10    1101    111010 ..... TCS  
 10    1110    111010 ..... TPOS  
 10    1111    111010 ..... TVC

op	rd	op3	rs1	1	simm13					
op	rd	op3	rs1	0	00000000				rs2	
31	29	24	18	13	12			4		0

10    00000    111011 ..... FLUSH  
 10                    111100 ..... SAVE  
 10                    111101 ..... RESTORE

op	rd	op4	rs1	1	simm13					
op	rd	op4	rs1	0	asi				rs2	
31	29	24	18	13	12			4		0

# Format 3

11	000000	LD
11	000001	LDUB
11	000010	LDUH
11	000011	LDD
11	000100	ST
11	000101	STB
11	000110	STH
11	000111	STD
11	001001	LDSB
11	001010	LDSH
11	001101	LDSTUB
11	001111	SWAP
11	010000	LDA†
11	010001	LDUBA†
11	010010	LDUHA†
11	010011	LDDA†
11	010100	STA†
11	010101	STBA†
11	010110	STHA†
11	010111	STDA†
11	011001	LDSBA†
11	011010	LDSHA†
11	011101	LDSTUBA†
11	011111	SWAPA†

op	rd	op4	rs1	1	simm13	
op	rd	op4	rs1	0	00000	rs2
31	29	24	18	13	12	4 0

11	100000	LDF
11	100001	LDFSR
11	100011	LDDF
11	100100	STF
11	100101	STFSR
11	100110	STDFQ†
11	100111	STDF

## ASI/Diagnostic Access

This appendix provides an overview of the SuperSPARC processor's (SSP's) ASI assignments, which are consistent with the *SPARC Reference MMU (SRMMU)* and the "Suggested ASI assignments" from the *SPARC Architecture Manual, version 8*.

Since SuperSPARC does not generally transmit ASI accesses external to the chip, system-visible ASI accesses are limited to:

- ☐ Transparent Memory Management Unit (MMU) mode (ASIs 0x20–0x2f)
- ☐ Normal data and instruction references (ASIs 0x08–0x0a)
- ☐ Control space accesses (ASI 0x02)

All eight bits of the ASI are decoded; an error occurs on any access to reserved ASI values. Each supported ASI specifies the types (LD/ST) and sizes of accesses allowed; violations cause errors.

0x0f	Data Cache Data	LD/ST	double	10.5.3
------	-----------------	-------	--------	--------

ASI 0x02 allows access to the information in an external device, nominally an external cache controller. The information may include cache controller (CC) registers, the external cache, and its directories. These accesses are non-cacheable, regardless of MCNTL.AC indication. Data references to this control space access may be any size. External system hardware is responsible for proper data alignment. Any faults will be reported as a `data_access_exception`.

Table B-1 lists all of the ASI values supported by the SSP. Each has been explained elsewhere in this manual. A reference to the manual section that describes the ASI is also provided.

### Note:

The instructions LDSTUBA and SWAPA generate `data_access_exception` on ASI values other than 0x08–0x0b or 0x20–0x2f. Any ASI access with a size other than indicated in the following table will generate a `data_access_exception`.

Table B-1. SuperSPARC Processor ASI Assignments

ASI	Function	Access	Size	Reference
0x00 to 0x01	reserved	-	-	-
0x02	Control Space Access	LD/ST	all	16.3
0x03	MMU Probe	LD/ST	single	9.8
0x04	MMU Registers	LD/ST	single	9.12
0x05	reserved	-	-	-
0x06	MMU TLB diagnostics	LD/ST	single	9.13
0x07	reserved	-	-	-
0x08	User Instruction	LD/ST	all	8.3
0x09	Supervisor Instruction	LD/ST	all	8.3
0x0a	User Data	LD/ST	all	8.3
0x0b	Supervisor Data	LD/ST	all	8.3
0x0c	Instruction Cache Tags	LD/ST	double	10.3.2
0x0d	Instruction Cache Data	LD/ST	double	10.3.3
0x0e	Data Cache Tags	LD/ST	double	10.5.2
0x10 to 0x1f	reserved	-	-	-
0x20 to 0x2f	MMU Bypass	LD/ST	all	9.9
0x30	Store Buffer Tags	LD/ST	double	10.7.1
0x31	Store Buffer Data	LD/ST	double	10.7.2
0x32	Store Buffer Control	LD/ST	single	10.7.3
0x33 to 0x35	reserved	-	-	-
0x36	Instruction Cache Flash clear	ST	single	10.3.1
0x37	Data cache flash clear	ST	single	10.5.1
0x38	MMU Breakpoint Diagnostics	LD/ST	double	15.2.1
0x39	BIST diagnostics	LD/ST	single	13.4.4
0x3a to 0x3f	reserved	-	-	-
0x40 to 0x41	Emulation Temps	LD/ST	single	22.4.2
0x42 to 0x43	reserved	-	-	-
0x44	Emulation Data In 1	LD	single	22.4.3
0x45	reserved	-	-	-
0x46	Emulation Data Out	LD/ST	single	22.4.4
0x47	Emulation Exit PC	LD/ST	single	22.4.1
0x48	Emulation Exit nPC	LD/ST	single	22.4.1
0x49	Emulation Counter Value	LD/ST	single	15.2.2
0x4a	Emulation Counter Control	LD/ST	single	15.2.3
0x4b	Emulation Counter Status	LD/ST	single	15.2.4
0x4c	Action Register	LD/ST	single	15.2.5
0x4d to 0xff	reserved	-	-	-

### SuperSPARC Processor Pin Description Tables

---

---

Table C-1, Table C-2, and Table C-3 list each pin of the SuperSPARC processor (SSP) and describe its function. Use Table C-1 for VBus configurations and Table C-2 for MBus configurations. Table C-3 lists power connections.

Table C-1. Pin Functions — VBus Interface ( $\overline{CCMODE} = L$ )

SIGNAL	I/O	PIN NO.	DESCRIPTION
ADDR35	I/O	A21	Physical Address Bus.
ADDR34		C21	
ADDR33		G21	
ADDR32		E21	
ADDR31		A19	
ADDR30		C19	
ADDR29		E19	
ADDR28		G19	
ADDR27		A17	
ADDR26		C17	
ADDR25		G17	
ADDR24		E17	
ADDR23		AR25	
ADDR22		AJ23	
ADDR21		AL23	
ADDR20		AN23	
ADDR19		AR23	
ADDR18		AJ21	
ADDR17		AL21	
ADDR16		AN21	
ADDR15		AR21	
ADDR14		AR19	
ADDR13		AN19	
ADDR12		AL19	
ADDR11		AJ19	
ADDR10		AR17	
ADDR09		AJ17	
ADDR08		AL17	
ADDR07		AN17	
ADDR06		AR15	
ADDR05		AN15	
ADDR04		AL15	
ADDR03		AJ15	
ADDR02		AR13	
ADDR01		AN13	
ADDR00		AL13	

Table C-1. Pin Functions — VBus Interface ( $\overline{CCMODE} = L$ ) (Continued)

ARDY	I	AL9	This signal is an input to indicate that system logic is prepared to accept another address or bus cycle. This signal needs to be low when using VBus interface. H = System not ready. L = System ready.
BURST	O	AL11	This signal is used to indicate that the current address on the bus is part of a burst bus cycle. H = Part of multi-cycle burst. L = Not part of multi-cycle burst.
BUSREQ	O	AJ9	Bus request. H = VBus not requested. L = VBus requested.
CCHBL	O	AL25	This signal indicates that the current transaction is internally cacheable. H = Noncacheable transaction. L = Cacheable transaction.
CCMODE	†	B26	Cache Controller Mode. Selects the operation of the SSP for standalone operation, or for operation with a cache controller (such as the MXCC). The operation of the store buffer, data cache operation and the bus interface (VBus or MBus) are selected from this signal. This signal must be statically asserted and not changed during normal operation. H = MBus interface of operation is selected, data cache operates copy-back. L = VBus interface of operation is selected, data cache operates write-through.
CMD $\overline{S}$	I/O	AJ5	Command strobe. Indicates the beginning of a bus cycle. When the SSP is not in bus master mode, as indicated by $\overline{WGRT}$ and $\overline{RGRT}$ being deasserted, $\overline{CMD\overline{S}}$ is used as a input to initiate external snoop transactions (including invalidates and demaps). H = Not a command word. L = VBus command word on ADDR35-ADDR00, CCHBL, CSA, DMAP, LDST, SIZE1-SIZE0, SU, RD, and WR.  When the SSP is a bus master it asserts this signal for the first cycle of a VBus transactions. H = Not a command word. L = VBus command word on ADDR35-ADDR00, CCHBL, CSA, DMAP, LDST, SIZE1-SIZE0, SU, RD, and WR.
CSA	O	AE7	This signal indicates that the current bus transaction is a control space access. It is asserted for the alternate space indicator (ASI) transactions to ASI space 0x02. H = normal memory or ASI access. L = control space access (to ASI 0x02).

† These pins are pulled inactive with weak internal resistive pull-ups.



Table C-1. Pin Functions — VBus Interface ( $\overline{CCMODE} = L$ ) (Continued)

SIGNAL	I/O	PIN NO.	DESCRIPTION
DATA63	I/O	E27	Data Bus.
DATA62		G27	
DATA61		F28	
DATA60		E29	
DATA59		G31	
DATA58		H30	
DATA57		J29	
DATA56		J31	
DATA55		J33	
DATA54		K34	
DATA53		L29	
DATA52		L31	
DATA51		L33	
DATA50		L35	
DATA49		N29	
DATA48		N31	
DATA47		N33	
DATA46		N35	
DATA45		R29	
DATA44		R33	
DATA43		U29	
DATA42		W35	
DATA41		W33	
DATA40		W29	
DATA39		W31	
DATA38		AA35	
DATA37		AA29	
DATA36		AA33	
DATA35		AA31	
DATA34		AC35	
DATA33		AC31	
DATA32		AC33	
DATA31		F8	
DATA30		G9	
DATA29		G5	
DATA28		H6	
DATA27		J7	
DATA26		J5	
DATA25		J3	
DATA24		K2	

Table C-1. Pin Functions — VBus Interface (CCMODE = L) (Continued)

SIGNAL	I/O	PIN NO.	DESCRIPTION
DATA23 DATA22 DATA21 DATA20 DATA19 DATA18 DATA17 DATA16 DATA15 DATA14 DATA13 DATA12 DATA11 DATA10 DATA09 DATA08 DATA07 DATA06 DATA05 DATA04 DATA03 DATA02 DATA01 DATA00	I/O	L7 L5 L3 L1 N5 N7 N3 N1 R1 R3 R7 R5 U1 U3 U5 U7 W1 W3 W7 W5 AA1 AA3 AA5 AC1	Data Bus.
DEMAPP	I/O	AG5	Asserted with CMDS to indicate demap cycle. As an input indicates an external demap cycle. When output: H = normal command word. L = demap cycle to system (System should remove TBL entries matching request). When input: H = non-demap cycle. L = demap cycle from system. The TBL entries matching request will be removed.
DPAR0 DPAR1 DPAR2 DPAR3 DPAR4 DPAR5 DPAR6 DPAR7	I/O†	AG31 AG33 AE29 AF34 AF2 AE5 AC7 AE3	Data bus parity. When parity is enabled (by setting the parity enable bits in the MCNTL register), even parity is generated and checked. When parity is disabled, odd parity is generated but parity is not checked. DPAR0 is parity for bits DATA63–DATA56, etc., as listed: DPAR0 – DATA63–DATA56      DPAR1 – DATA55–DATA48 DPAR2 – DATA47–DATA40      DPAR3 – DATA39–DATA32 DPAR4 – DATA31–DATA24      DPAR5 – DATA23–DATA16 DPAR6 – DATA15–DATA08      DPAR7 – DATA07–DATA00
ERROR	O	AJ25	This signal indicates that the SSP has entered an error mode state and will take a <i>watchdog</i> reset trap. H = Normal operation. L = Error Mode.
ESB	O	C13	Execution strobe output. H = Programmed breakpoint event is occurring. L = Inactive.

† These pins are pulled inactive with weak internal resistive pull-ups.

Table C-1. Pin Functions — VBus Interface ( $CCMODE = L$ ) (Continued)

SIGNAL	I/O	PIN NO.	DESCRIPTION																																				
IRL3 IRL2 IRL1 IRL0	I	A15 C15 E15 A13	Interrupt request level. This field specifies the level of the highest priority interrupt request that is currently pending. If IRL3 – IRL0 = 0000, no interrupts are pending. Level 15 (IRL3 – IRL0 = 1111) is a NMI (disable all traps) Level 14 Highest Maskable Interrupt Level 1 Lowest Maskable Interrupt Level 0 No Interrupts are Pending																																				
LDST	O	AG3	This signal indicates an atomic load/store (LDSTUB, LDSTUBA, SWAP, or SWAPA) operation. It is equivalent to the logical OR of RD and WR signals. H = No LDST. L = Atomic Load/Store (LDST) cycle.																																				
MEXC	I	AJ11	This signal is encoded with <b>RDY</b> or <b>WRDY</b> and with <b>RETRY</b> to indicate the type of acknowledgment. <table> <thead> <tr> <th>MEXC</th><th>RDY/WRDY</th><th>RETRY</th><th>Description</th></tr> </thead> <tbody> <tr> <td>1</td><td>1</td><td>1</td><td>No reply</td></tr> <tr> <td>1</td><td>1</td><td>0</td><td>Retry</td></tr> <tr> <td>1</td><td>0</td><td>1</td><td>Data transfer complete</td></tr> <tr> <td>1</td><td>0</td><td>0</td><td>Undefined error (UD)</td></tr> <tr> <td>0</td><td>1</td><td>1</td><td>Bus error (BE)</td></tr> <tr> <td>0</td><td>1</td><td>0</td><td>Timeout error (TO)</td></tr> <tr> <td>0</td><td>0</td><td>1</td><td>Reserved</td></tr> <tr> <td>0</td><td>0</td><td>0</td><td>Reserved</td></tr> </tbody> </table>	MEXC	RDY/WRDY	RETRY	Description	1	1	1	No reply	1	1	0	Retry	1	0	1	Data transfer complete	1	0	0	Undefined error (UD)	0	1	1	Bus error (BE)	0	1	0	Timeout error (TO)	0	0	1	Reserved	0	0	0	Reserved
MEXC	RDY/WRDY	RETRY	Description																																				
1	1	1	No reply																																				
1	1	0	Retry																																				
1	0	1	Data transfer complete																																				
1	0	0	Undefined error (UD)																																				
0	1	1	Bus error (BE)																																				
0	1	0	Timeout error (TO)																																				
0	0	1	Reserved																																				
0	0	0	Reserved																																				
OE	I/O	AR11	As an output, this signal controls the pipelined output enable of external cache SRAM. It is used as an input to prevent bus collisions. H = SRAM outputs disabled. L = SRAM outputs enabled.																																				
PEND	I	AJ31	This signal indicates that at least one outstanding write operation has not completed. H = System has no incompleted write operations outstanding from this processor. L = System has write operations that were issued by this processor that are not yet complete.																																				
PIPE9	O	E13	H = A valid memory reference occurred in the E0 stage of the previous clock cycle. L = No valid memory reference occurred in the E0 stage of the previous clock cycle.																																				
PIPE8	O	A11	H = A valid floating point operation occurred in the E0 stage of the previous clock cycle. L = No valid floating point operation occurred in the E0 stage of the previous clock cycle.																																				
PIPE7	O	C11	H = A valid control transfer instruction was executed in the E0 stage of the previous clock cycle. L = No valid control transfer instruction was executed in the E0 stage of the previous clock cycle.																																				
PIPE6	O	G13	H = Indicates that no instructions were available when the group currently at the WB stage was in the D0 stage. L = Indicates that one or more instructions were available in this group.																																				
PIPE5	O	B10	H = The pipeline is being held by the data cache (generally processing a cache miss). L = The pipeline is not being held by the data cache.																																				
PIPE4	O	E11	H = The pipeline is being held by the FPU (either queue is full or dependencies). L = The pipeline is not being held by the FPU.																																				
PIPE3	O	C9	H = Indicates that the branch in E0 stage of the previous cycle was taken. L = Indicates that the branch in E0 stage of the previous cycle was not taken.																																				

Table C-1. Pin Functions — VBus Interface (CCMODE = L) (Continued)

SIGNAL	I/O	PIN NO.	DESCRIPTION										
PIPE2 PIPE1	O	G11 E9	Indicates the number of instructions in the E0 stage of the current cycle:  <table><tr><th>PIPE2 – PIPE1</th><th>Instructions in E0 Stage</th></tr><tr><td>00</td><td>None</td></tr><tr><td>01</td><td>1</td></tr><tr><td>10</td><td>2</td></tr><tr><td>11</td><td>3</td></tr></table>	PIPE2 – PIPE1	Instructions in E0 Stage	00	None	01	1	10	2	11	3
PIPE2 – PIPE1	Instructions in E0 Stage												
00	None												
01	1												
10	2												
11	3												
PIPE0	O	E7	H = Indicates that there is an exception or interrupt being signalled in the current cycle. L = Indicates that there is not an exception or interrupt being signalled in the current cycle.										
PLLBYP	†	U33	This pin is used to bypass the internal phase lock loop. When this pin is asserted, the external clock input will be routed directly to internal clock distribution with no delay compensation. H = PLL enabled. Normal operation. L = PLL disabled. No clock delay compensation.										
RD	I/O	AN11	SSP drives RD to qualify addresses on the VBus as READ cycles. It is also asserted with WR for swap cycles and with DEMAP for demap cycles. As an input, used for internal SRAM test only. H = Not a read cycle. L = Read (or Load/Store with WR and LDST low) cycle.										
RESET	I	G25	Reset. This causes an external reset for the SSP. At power-on, RESET must be held low for at least 100 ms to allow the PLL to stabilize. If the PLL is known to be stable, RESET may be asserted for as short as 8 cycles. See reset operation. H = Normal operation. L = The SSP is externally reset.										
RETRY	I	AP10	This signal is encoded along with RRDY or WRDY and with MEXC to indicate the type of acknowledgment. See MEXC description for table.										
RGRT	I	AH6	This signal indicates that the SSP has been given a grant to use the VBus for read operations. H = VBus not available for read operations. L = VBus available for read operations.										
RRDY	I	AL7	This signal indicates that incoming read data is valid. RRDY may be connected to WRDY when only a single ready signal is required. This signal is encoded with MEXC and RETRY. See MEXC description for table.										
SIZE1 SIZE0	O	AJ27 AK28	These bits indicate the transfer size of the current transaction. 00 = Byte 01 = Half Word 10 = Word 11 = Doubleword										
spare3	O	AH30	Not used. Should be tied high or left floating during normal chip operation.										
spare2 spare1 spare0	I	A25 C27 AG29	Not used. Should be tied high or left floating during normal chip operation.										
SRMTST	†	C25	Reserved: Factory test pin. It must be connected to V <sub>CC</sub> for normal operation.										
SU	O	AN25	This signal indicates that the current bus transaction is a supervisor transaction. H = User (unprivileged) transaction L = Supervisor (privileged) transaction										

† These pins are pulled inactive with weak internal resistive pull-ups.

Table C-1. Pin Functions — VBus Interface (*CCMODE* = L) (Continued)

SIGNAL	I/O	PIN NO.	DESCRIPTION
TCK	I†	G23	JTAG test clock input.
TDI	I†	E23	JTAG test data input.
TDO	O	G15	JTAG test data output.
TEST	I†	E25	This pin can be used for board level testing. H = Normal operation. L = All outputs except ESB and TDO are placed in a high-impedance state.
TMS	I†	G23	JTAG test mode select input.
TRST	I†	A23	JTAG test reset input.
VCLK	I	R35	Primary clock source.
VPLLRC	I	U35	Phase Locked Loop filter capacitor. This pin should be connected to an external 0.1 $\mu$ F capacitor to ground.
WE0 WE1 WE2 WE3 WE4 WE5 WE6 WE7	O	AE31 AE33 AE35 AC29 AE1 AC5 AC3 AA7	These signals directly control the write enable signals of synchronous SRAM used for external cache. These signals are driven only when asserted; otherwise, they are three-state. WE bit ordering corresponds to the <i>big-endian</i> convention (i.e., WE0 is the write enable for byte 0 (DATA63 – DATA56)). H = SRAM read. L = SRAM write.
WEE	I	AN9	This pin is used to control the assertion of WE7 – WE0 signals. H = May not drive WE7 – WE0. L = May drive WE7 – WE0.
WR	I/O	AJ13	SSP drives WR to qualify addresses on the bus as write cycle. It is asserted with RD for swaps as well as demap cycles. As an input, this signal is used to qualify invalidation requests. H = Not a write cycle. L = Write (or Load/Store with RD and LDST low) cycle.
WRDY	I	AK8	This signal indicates that incoming read data is valid. WRDY may be connected to RRDY when only a single ready signal is required. This signal is encoded with MEXC and RETRY. See MEXC description for table.
WGRT	I	AG7	This signal grants the SSP bus access for write operations. WGRT may be connected to RGRT when only a single grant line is required. H = VBus not available for write operations. L = VBus available for write operations.
nu	†	AN27	Not used in the VBus interface.
nu	†	AP26	Not used in the VBus interface.

† These pins are pulled inactive with weak internal resistive pull-ups.

Table C-2. Pin Functions — MBus Interface (*CCMODE* = H)

SIGNAL	I/O	PIN NO	DESCRIPTION
AERR	O <sup>‡</sup>	AJ25	In error mode, the SSP will perform an automatic watchdog reset. Error mode is entered when any exception is taken with traps disabled (PSR.ET=0). This signal is driven only when asserted; otherwise, it is three-state. H = Normal operation. L = Error mode.
CCMODE	I <sup>†</sup>	B26	Cache Controller Mode. Selects the operation of the SSP for standalone operation, or for operation with a cache controller (such as the MXCC). The operation of the store buffer, data cache operation and the bus interface (VBus or MBus) are selected from this signal. This signal must be statically asserted and not changed during normal operation. H = MBus interface of operation is selected, data cache operates copy-back. L = VBus interface of operation is selected, data cache operates write-through.
CLK	I	R35	Primary clock source.
ESB	O	C13	Execution strobe output. H = Programmed breakpoint event is occurring. L = Inactive.

† These pins are pulled inactive with weak internal resistive pull-ups.

‡ These pins have an open drain.

Table C-2. Pin Functions — MBus Interface (CCMODE = H) (Continued)

MAD63		E27	
MAD62		G27	
MAD61		F28	
MAD60		E29	
MAD59		G31	
MAD58		H30	
MAD57		J29	
MAD56		J31	
MAD55		J33	
MAD54		K34	
MAD53		L29	
MAD52		L31	
MAD51		L33	
MAD50		L35	
MAD49		N29	
MAD48		N31	
MAD47		N33	
MAD46		N35	
MAD45		R29	
MAD44	I/O	R33	Multiplexed Command / Data.
MAD43		U29	
MAD42		W35	
MAD41		W33	
MAD40		W29	
MAD39		W31	
MAD38		AA35	
MAD37		AA29	
MAD36		AA33	
MAD35		AA31	
MAD34		AC35	
MAD33		AC31	
MAD32		AC33	
MAD31		F8	
MAD30		G9	
MAD29		G5	
MAD28		H6	
MAD27		J7	
MAD26		J5	
MAD25		J3	

Table C-2. Pin Functions — MBus Interface (*CCMODE* = H) (Continued)

SIGNAL	I/O	PIN NO	DESCRIPTION
MAD24 MAD23 MAD22 MAD21 MAD20 MAD19 MAD18 MAD17 MAD16 MAD15 MAD14 MAD13 MAD12 MAD11 MAD10 MAD09 MAD08 MAD07 MAD06 MAD05 MAD04 MAD03 MAD02 MAD01 MAD00	I/O	K2 L7 L5 L3 L1 N5 N7 N3 N1 R1 R3 R7 R5 U1 U3 U5 U7 W1 W3 W7 W5 AA1 AA3 AA5 AC1	Multiplexed Command / Data.
MAS	I/O	AL9	MBus Address Strobe. Asserted by the bus master when an MBus command word (containing address and control information) is on MAD63 – MAD0. H = No command word. L = MBus command word on MAD63 – MAD0.
MBB	I/O	AN9	MBus Busy. Asserted when there is any active transaction on MBus. H = MBus free. L = MBus busy.
MBG	I	AG7	MBus Grant. This is a dedicated (not bused) signal from the MBus arbiter to this bus master. H = Not granted. The SSP may not initiate an MBus transaction. L = Granted. The SSP may initiate an MBus transaction as soon as MBus is free.
MBR	O	AJ9	MBus Request. This is a dedicated (not bused) signal from this bus master to the MBus arbiter. H = No request. L = Requesting to initiate a transaction on MBus.



Table C-2. Pin Functions — MBus Interface (CCMODE = H) (Continued)

SIGNAL	I/O	PIN NO	DESCRIPTION			
MERR	I	AJ11	MBus Error. Encoded along with MRDY and MRTY to indicate acknowledgment type (the type of error response).			
			MERR	MRDY	MRTY	Description
			H	H	H	Idle cycle
			H	H	L	Relinquish and retry
			H	L	H	Valid data transfer
			H	L	L	Reserved
			L	H	H	Bus error (ERROR1)
			L	H	L	Timeout error (ERROR2)
			L	L	H	Uncorrectable error (ERROR3)
L	L	L	Retry			
MID3 MID2 MID1 MID0	I†	AJ15 AR13 AN13 AL13	MBus Module ID. The identifier of this MBus device. Usually hardwired by the system. MID3 is the Most Significant bit (MSb) and MID0 is the Least Significant bit (LSb).			
MIF	I/O	AN27	Memory Inhibit. Asserted by a snooping cache when it notices a coherent read of a cache block it owns. Memory responds to this signal by ignoring the request. H = No memory inhibit. L = Inhibit memory. The snooping cache which asserted MIF will respond with the data in place of memory.			
MIRL3 MIRL2 MIRL1 MIRL0	I	A15 C15 E15 A13	Interrupt Request Level. This field specifies the level of the highest priority interrupt request that is currently pending. If MIRL3 – MIRL0 = 0000, no interrupts are pending. Level 15 (MIRL3 – MIRL0 = 1111) is a NMI (disable all traps)      Level 14 Highest Maskable Interrupt Level 1 Lowest Maskable Interrupt      Level 0 No Interrupts are Pending			
MRDY	I/O	AK8	MBus Ready. Encoded along with MERR and MRTY to indicate acknowledgment type (the type of error response). See table in MERR description.			
MRTY	I	AP10	MBus Retry. Encoded along with MERR and MRDY to indicate acknowledgment type (the type of error response). See table in MERR description.			
MSH	I/O‡	AP26	Memory Shared. Asserted by a snooping cache when it notices a coherent read of a cache block it is caching. Both caches will mark the data as shared. H = No sharing. L = Shared data.			
PEND	I†	AJ31	This signal is generally used in the VBus interface only. It indicates to the SSP that at least one outstanding write operation has not completed. H = System has no write operations outstanding from this processor. L = System has write operations that were issued by this processor that are not yet complete.			
PIPE9	O	E13	H = A valid memory reference occurred in the E0 stage of the previous clock cycle. L = No valid memory reference occurred in the E0 stage of the previous clock cycle.			
PIPE8	O	A11	H = A valid floating point operation occurred in the E0 stage of the previous clock cycle. L = No valid floating point operation occurred in the E0 stage of the previous clock cycle.			

† These pins are pulled inactive with weak internal resistive pull-ups.

‡ These pins have an open drain.

Table C-2. Pin Functions — MBus Interface (CCMODE = H) (Continued)

SIGNAL	I/O	PIN NO	DESCRIPTION										
PIPE7	O	C11	H = A valid control transfer instruction was executed in the E0 stage of the previous clock cycle. L = No valid control transfer instruction was executed in the E0 stage of the previous clock cycle.										
PIPE6	O	G13	H = Indicates that no instructions were available when the group currently at the WB stage was in the D0 stage. L = Indicates that one or more instructions were available in this group.										
PIPE5	O	B10	H = The pipeline is being held by the data cache (generally processing a cache miss). L = The pipeline is not being held by the data cache.										
PIPE4	O	E11	H = The pipeline is being held by the FPU (either queue is full or dependencies). L = The pipeline is not being held by the FPU.										
PIPE3	O	C9	H = Indicates that the branch in E0 stage of the previous cycle was taken. L = Indicates that the branch in E0 stage of the previous cycle was not taken.										
PIPE2 PIPE1	O	G11 E9	Indicates the number of instructions in the E0 stage of the current cycle: <table><tr><th>PIPE2-PIPE1</th><th>Instructions in E0 Stage</th></tr><tr><td>00</td><td>None</td></tr><tr><td>01</td><td>1</td></tr><tr><td>10</td><td>2</td></tr><tr><td>11</td><td>3</td></tr></table>	PIPE2-PIPE1	Instructions in E0 Stage	00	None	01	1	10	2	11	3
PIPE2-PIPE1	Instructions in E0 Stage												
00	None												
01	1												
10	2												
11	3												
PIPE0	O	E7	H = Indicates that there is an exception or interrupt being signalled in the current cycle. L = Indicates that there is not an exception or interrupt being signalled in the current cycle.										
PLLBYP	I†	U33	This pin is used to bypass the internal phase lock loop. When this pin is asserted, the external clock input will be routed directly to internal clock distribution with no delay compensation. H = PLL enabled. Normal operation. L = PLL disabled. No clock delay compensation.										
RSTIN	I	G25	Reset In. This causes an external reset for the SSP. At power-on, RSTIN must be held low for at least 100 ms to allow the PLL to stabilize. If the PLL is known to be stable, RSTIN may be asserted for as short as 8 cycles. See reset operation H = Normal operation. L = The SSP is externally reset.										
spare3	O	AH30	Not used. Should be tied high or left floating during normal chip operation.										
spare2 spare1 spare0	I	A25 C27 AG29	Not used. Should be tied high or left floating during normal chip operation.										
TCK	I†	G23	JTAG test clock input.										
TDI	I†	E23	JTAG test data input.										
TDO	O	G15	JTAG test data output.										
TEST	I†	E25	This pin can be used for board level testing. H = Normal operation. L = All outputs except ESB and TDO are placed in a high-impedance state.										
TMS	I†	C23	JTAG test mode select input.										

† These pins are pulled inactive with weak internal resistive pull-ups.

Table C-2. Pin Functions — MBus Interface (CCMODE = H) (Continued)

SIGNAL	I/O	PIN NO	DESCRIPTION
TRST	I	A23	JTAG test reset input.
VPLLRC	I	U35	Phase locked loop filter capacitor. This pin should be connected to an external 0.1 $\mu$ F capacitor to ground.
nu		AJ27 AK28	Not used for MBus.
nu		AE31 AE33 AE35 AC29 AE1 AC5 AC3 AA7	Not used for MBus.
nu		AN25	Not used for MBus.
nu		AL11	Not used for MBus.
nu	†	AG5	Not used for MBus.
nu		AG31 AG33 AE29 AF34 AF2 AE5 AC7 AE3	Not used for MBus.
nu	†	AN11	Not used for MBus.
nu	†	C25	Not used for MBus.
nu	†	AJ13	Not used for MBus.
nu	†	AJ5	Not used for MBus.
nu	†	AR11	Not used for MBus.
nu	†	AH6	Not used for MBus.
nu	†	AL7	Not used for MBus.
nu		AL25	Not used for MBus.
nu		AE7	Not used for MBus.
nu		AG3	Not used for MBus.

† These pins are pulled inactive with weak internal resistive pull-ups.

Table C-2. Pin Functions — MBus Interface ( $\overline{CCMODE} = H$ ) (Continued)

SIGNAL	I/O	PIN NO	DESCRIPTION
nu		A21	Not used for MBus.
		C21	
		G21	
		E21	
		A19	
		C19	
		E19	
		G19	
		A17	
		C17	
		G17	
		E17	
		AR25	
		AJ23	
		AL23	
		AN23	
		AR23	
		AJ21	
		AL21	
		AN21	
		AR21	
		AR19	
		AN19	
		AL19	
		AJ19	
		AR17	
		AJ17	
		AL17	
		AN17	
		AR15	
		AN15	
		AL15	

† These pins are pulled inactive with weak internal resistive pull-ups.

Table C-3. Pin Functions — Power Connections

SIGNAL	I/O	PIN NUMBER	DESCRIPTION
V <sub>CC</sub>	I	K4, P4, AB4, AF4, AM10, AM14, AM22, AM26, AF32, AB32, P32, K32, D26, D22, D14, D10	+5 volts for core logic.
V <sub>CCCLK</sub> <sup>†</sup>	I	R31	+5 volts for clock and PLL.
V <sub>CCI</sub>	I	F6, V2, AK6, AP18, AK30, V34, F30, B18	+5 volts for input buffers.
V <sub>CCP</sub>	I	M6, P2, T6, Y6, AB2, AD6, AK12, AP14, AK16, AK20, AP22, AK24, AD30, AB34, Y30, T30, P34, M30, F24, B22, F20, F16, B14, F12	+5 volts for peripheral logic.
V <sub>SS</sub>	I	H4, M4, T4, Y4, AD4, AH4, AM8, AM12, AM16, AM20, AM24, AM28, AH32, AD32, Y32, T32, M32, H32, D28, D24, D20, D16, D12, D8	Ground for core logic.
V <sub>SSCLK</sub>	I	U31	Ground for clock and PLL.
V <sub>SSI</sub>	I	G7, V4, AJ7, AM18, AJ29, V32, G29, D18	Ground for input buffers.
V <sub>SSP</sub>	I	K6, M2, P6, T2, V6, Y2, AB6, AD2, AF6, AK10, AP12, AK14, AP16, AK18, AP20, AK22, AP24, AK26, AF30, AD34, AB30, Y34, V30, T34, P30, M34, K30, F26, B24, F22, B20, F18, B16, F14, B12, F10	Ground for peripheral logic.

### MultiCache Controller Pin Description Tables

---

---

---

Table D-1, Table D-2, and Table D-3 list each pin of the MultiCache Controller (MXCC) and describe its function. Use Table D-1 for configurations of the MXCC using MBus and Table D-2 for configurations using XBus. Table D-3 lists power connections.

Table D-1. Pin Functions — MBus Configuration (MBSEL = H)

PIN NAME	I/O	PIN NO.	FUNCTION
ADDR35	I/O†	V32	Processor physical address bus.
ADDR34		Y34	
ADDR33		Y32	
ADDR32		AA35	
ADDR31		Y28	
ADDR30		AB34	
ADDR29		AA29	
ADDR28		AA31	
ADDR27		AB32	
ADDR26		AC35	
ADDR25		AD34	
ADDR24		AC31	
ADDR23		E21	
ADDR22		B24	
ADDR21		H22	
ADDR20		E23	
ADDR19		A23	
ADDR18		D22	
ADDR17		G21	
ADDR16		B22	
ADDR15		D20	
ADDR14		H20	
ADDR13		A21	
ADDR12		B20	
ADDR11		G19	
ADDR10		E19	
ADDR09		H18	
ADDR08		B18	
ADDR07		D18	
ADDR06		E17	
ADDR05		G17	
ADDR04		D16	
ADDR03		B16	
ADDR02		H16	
ADDR01		A15	
ADDR00		G15	

† These pins have internal holding drivers.

Table D-1. Pin Functions — MBus Configuration (MBSEL = H) (Continued)

PIN NAME	I/O	PIN NO.	FUNCTION
AERR	O	AC5	Indicates either an internal MXCC error or ERROR is asserted by the processor. H = No error L = An internal processor or MXCC error
BPLLRC	I	AE31	Capacitor for the phase filter of the bus clock PLL. This pin should be connected to an external capacitor to ground. With an internal resistor, this circuit provides the RC time constant for the phase filter of the bus clock domain PLL.
BURST	†	H14	Indicates whether a burst access is in progress. BURST is driven at the same time as ADDR35 – ADDR0, and it is asserted during both read bursts and write bursts. BURST is deasserted on the last address of a burst to allow the MXCC to stop returning RRDY or WRDY with the last data of the burst. H = A burst access is in progress. L = A burst access is not in progress.
CCHBL	†	D24	Cacheable access. This pin indicates the current processor transaction as one that may be cached in an external cache. H = Noncacheable access. L = Cacheable access.
CMDS	I/O†	A9	Command strobe. Indicates the beginning of a bus cycle. The VBus master asserts this signal for one cycle to begin all of its accesses. When the MXCC is a bus master, as indicated by $\overline{WGRT}$ and $\overline{RGRT}$ being deasserted, it asserts CMDS to initiate invalidate and demap transactions. H = Not a command word L = VBus invalidate or demap command word on ADDR35 – ADDR0, DEMAP, and WR.  When the MXCC is not a bus master, this signal indicates the first cycle of a VBus transaction. H = Not a command word. L = VBus command word on ADDR35 – ADDR0, CCHBL, CSA, DEMAP, IDST, SIZE1 – SIZE0, SU, RD, and WR.
CSA	†	G11	Control-space access. The processor asserts this signal when performing a read or write to the internal tag RAM, E-cache, or registers of the MXCC. H = Normal memory access. L = Control-space access.

† These pins have internal holding drivers.

‡ These pins have internal pullup resistors.



Table D-1. Pin Functions — MBus Configuration (MBSEL = H) (Continued)

PIN NAME	I/O	PIN NO.	FUNCTION
DATA63		U31	
DATA62		U29	
DATA61		T34	
DATA60		T32	
DATA59		T28	
DATA58		R35	
DATA57		R29	
DATA56		P32	
DATA55		R31	
DATA54		N35	
DATA53		P28	
DATA52		M34	
DATA51		P34	
DATA50		N29	
DATA49		M32	
DATA48		L35	
DATA47		N31	
DATA46		L31	
DATA45		M28	
DATA44	I/O†	L29	Processor data bus.
DATA43		K34	
DATA42		K32	
DATA41		J35	
DATA40		H34	
DATA39		J31	
DATA38		K28	
DATA37		J29	
DATA36		H32	
DATA35		G31	
DATA34		F32	
DATA33		H28	
DATA32		D28	
DATA31		V4	
DATA30		V8	
DATA29		U5	
DATA28		U7	
DATA27		T2	
DATA26		T4	
DATA25		R1	

† These pins have internal holding drivers.

‡ These pins have internal pullup resistors.

Table D-1. Pin Functions — MBus Configuration (MBSEL = H) (Continued)

PIN NAME	I/O	PIN NO.	FUNCTION
DATA24 DATA23 DATA22 DATA21 DATA20 DATA19 DATA18 DATA17 DATA16 DATA15 DATA14 DATA13 DATA12 DATA11 DATA10 DATA9 DATA8 DATA7 DATA6 DATA5 DATA4 DATA3 DATA2 DATA1 DATA0	I/O†	T8 R5 P2 R7 P4 N1 P8 N5 N7 L5 M2 M4 L1 L7 M8 K4 J1 K2 J5 H2 K8 H4 G5 J7 F4	Processor data bus (continued).
DEMAP	I/O†	E13	<p>INPUT: Asserted with a Demap Data Word on DATA63 – DATA0 and WR asserted to pass a demap request from the processor to the MXCC, and then to the system bus. DEMAP asserted with RD asserted indicates that the processor has successfully completed a demap operation requested by the MXCC (initiated from the system bus)</p> <p>H = No demap request L = When WR process has requested a demap cycle When RD process has completed a system bus requested demap cycle</p> <p>OUTPUT: Asserted when the system bus has requested a demap operation. DATA63 – DATA0 contains a Demap Data Word indicating which virtual address translations are to be discarded.</p> <p>H = No demap request L = System bus requested demap cycle</p>
DPAR0 DPAR1 DPAR2 DPAR3 DPAR4 DPAR5 DPAR6 DPAR7	I/O†	D26 A27 H26 G27 D10 E9 E7 H10	<p>Data bus parity. When parity is enabled, even parity is generated and checked. DPAR0 is parity for bits DATA63 – DATA56. When parity checking is disabled, odd parity is generated but not checked.</p> <p>DPAR0: DATA63 – DATA56      DPAR1: DATA55 – DATA48 DPAR2: DATA47 – DATA40      DPAR3: DATA39 – DATA32 DPAR4: DATA31 – DATA24      DPAR5: DATA23 – DATA16 DPAR6: DATA15 – DATA08      DPAR7: DATA07 – DATA00</p>
ERROR	†	A25	<p>Processor error. The processor asserts this pin when it has entered an internal error state. The MXCC initiates an internal reset when ERROR is asserted.</p> <p>H = Normal operation L = Processor internal error</p>

† These pins have internal holding drivers.

‡ These pins have internal pullup resistors.

Table D-1. Pin Functions — MBus Configuration (MBSEL = H) (Continued)

PIN NAME	I/O	PIN NO.	FUNCTION
IRL3 IRL2 IRL1 IRL0	O	AB28 AC29 AD32 AE29	Interrupt request level. This field specifies to the processor the level of the highest priority interrupt request that is currently pending. If IRL3 – IRL0 = 0000, no interrupts are pending. Level 15 (IRL3 – IRL0 = 1111): Nonmaskable interrupt. Level 14: Highest maskable interrupt. Level 1: Lowest maskable interrupt. Level 0: No interrupts are pending
LDST	†	B10	This signal indicates an atomic load/store (LDSTUB, LDSTUBA, SWAP, or SWAPA) operation. It is equivalent to the logical OR of RD and WR signals. No other transactions may occur while LDST is asserted. H = No LDST L = Atomic load/store (LDST) cycle

† These pins have internal holding drivers.

‡ These pins have internal pullup resistors.

Table D-1. Pin Functions — MBus Configuration (MBSEL = H) (Continued)

MAD63		AF32	
MAD62		AG35	
MAD61		AG31	
MAD60		AH34	
MAD59		AH32	
MAD58		AG29	
MAD57		AJ31	
MAD56		AK32	
MAD55		AH28	
MAD54		AM30	
MAD53		AL29	
MAD52		AM28	
MAD51		AJ27	
MAD50		AP28	
MAD49		AH26	
MAD48		AL27	
MAD47		AR27	
MAD46		AM26	
MAD45		AP26	
MAD44		AJ25	
MAD43		AH24	
MAD42		AR25	
MAD41		AM24	
MAD40		AP24	
MAD39		AL25	
MAD38	I/O	AL23	MBus multiplexed command / data bus.
MAD37		AH22	
MAD36		AR23	
MAD35		AM22	
MAD34		AP22	
MAD33		AL21	
MAD32		AJ21	
MAD31		AJ15	
MAD30		AP14	
MAD29		AM14	
MAD28		AR13	
MAD27		AL13	
MAD26		AH14	
MAD25		AP12	
MAD24		AJ13	
MAD23		AM12	
MAD22		AR11	
MAD21		AL11	
MAD20		AP10	
MAD19		AH12	
MAD18		AM10	
MAD17		AJ11	
MAD16		AR9	
MAD15		AP8	
MAD14		AM8	
MAD13		AH10	
MAD12		AJ9	

Table D-1. Pin Functions — MBus Configuration (MBSEL = H) (Continued)

PIN NAME	I/O	PIN NO.	FUNCTION																																				
MAD11 MAD10 MAD09 MAD08 MAD07 MAD06 MAD05 MAD04 MAD03 MAD02 MAD01 MAD00	I/O	AL9 AL7 AM6 AK4 AJ5 AG7 AF8 AG5 AH4 AH2 AG1 AF4	MBus multiplexed command / data bus (continued).																																				
MAS	I/O <sup>†</sup>	AJ17	MBus address strobe. Asserted by current master when a valid address/command is present on MAD63 – MAD0. H = A valid address/command is not present on MAD63 – MAD0 L = A valid address/command is present on MAD63 – MAD0																																				
MBB	I/O <sup>†</sup>	AM16	MBus busy. MBB is a 3-state signal that is asserted by the current bus master as long as the current bus master is using the MBus. Rising edge = Completed MBus transactions (released) Hi-Z = MBus not busy L = MBus is busy																																				
MBG	I	AL17	MBus grant. This is a nonbused signal from the MBus arbiter to a potential master. It is asserted by the external arbiter when this master has been granted the MBus. H = The MXCC not granted the MBus L = The MXCC granted the MBus																																				
MBR	O	AP20	MBus request. This is a nonbused signal to the MBus arbiter. It is asserted by the MXCC when it needs to access MBus. H = The MXCC does not need to access the MBus L = The MXCC needs to access the MBus																																				
MBSEL	I <sup>‡</sup>	V2	MBus select. This signal is used to select the system bus interface. This signal should not be changed during operation of this device. H = MBus system interface L = XBus system interface																																				
MCLK	I	AF34	Bus clock.																																				
MERR	I/O	AL15	MBus error. Encoded along with MRDY and MRTY to indicate acknowledgment type (the type of error response). <table> <tr> <th>MERR</th><th>MRDY</th><th>MRTY</th><th>Description</th></tr> <tr> <td>H</td><td>H</td><td>H</td><td>Idle cycle</td></tr> <tr> <td>H</td><td>H</td><td>L</td><td>Relinquish and retry</td></tr> <tr> <td>H</td><td>L</td><td>H</td><td>Valid data transfer</td></tr> <tr> <td>H</td><td>L</td><td>L</td><td>Reserved</td></tr> <tr> <td>L</td><td>H</td><td>H</td><td>Bus error (ERROR1)</td></tr> <tr> <td>L</td><td>H</td><td>L</td><td>Timeout error (ERROR2)</td></tr> <tr> <td>L</td><td>L</td><td>H</td><td>Uncorrectable error (ERROR3)</td></tr> <tr> <td>L</td><td>L</td><td>L</td><td>Retry</td></tr> </table>	MERR	MRDY	MRTY	Description	H	H	H	Idle cycle	H	H	L	Relinquish and retry	H	L	H	Valid data transfer	H	L	L	Reserved	L	H	H	Bus error (ERROR1)	L	H	L	Timeout error (ERROR2)	L	L	H	Uncorrectable error (ERROR3)	L	L	L	Retry
MERR	MRDY	MRTY	Description																																				
H	H	H	Idle cycle																																				
H	H	L	Relinquish and retry																																				
H	L	H	Valid data transfer																																				
H	L	L	Reserved																																				
L	H	H	Bus error (ERROR1)																																				
L	H	L	Timeout error (ERROR2)																																				
L	L	H	Uncorrectable error (ERROR3)																																				
L	L	L	Retry																																				

<sup>†</sup> These pins have internal holding drivers.<sup>‡</sup> These pins have internal pullup resistors.

Table D-1. Pin Functions — MBus Configuration (MBSEL = H) (Continued)

PIN NAME	I/O	PIN NO.	FUNCTION																																				
MEXC	O	G13	<p>Memory exception. This signal is asserted when the memory controller could not return or accept the requested data. This signal may cause the processor to take a memory exception trap. This signal is encoded with RRDY or WRDY, and RETRY to indicate the type of acknowledgment.</p> <table> <tr> <th>MEXC</th><th>RRDY/WRDY</th><th>RETRY</th><th>Description</th></tr> <tr> <td>H</td><td>H</td><td>H</td><td>No reply</td></tr> <tr> <td>H</td><td>H</td><td>L</td><td>Retry</td></tr> <tr> <td>H</td><td>L</td><td>H</td><td>Data transfer complete</td></tr> <tr> <td>H</td><td>L</td><td>L</td><td>Undefined error (UD)</td></tr> <tr> <td>L</td><td>H</td><td>H</td><td>Bus error (BE)</td></tr> <tr> <td>L</td><td>H</td><td>L</td><td>Timeout error (TO)</td></tr> <tr> <td>L</td><td>L</td><td>H</td><td>Reserved</td></tr> <tr> <td>L</td><td>L</td><td>L</td><td>Reserved</td></tr> </table>	MEXC	RRDY/WRDY	RETRY	Description	H	H	H	No reply	H	H	L	Retry	H	L	H	Data transfer complete	H	L	L	Undefined error (UD)	L	H	H	Bus error (BE)	L	H	L	Timeout error (TO)	L	L	H	Reserved	L	L	L	Reserved
MEXC	RRDY/WRDY	RETRY	Description																																				
H	H	H	No reply																																				
H	H	L	Retry																																				
H	L	H	Data transfer complete																																				
H	L	L	Undefined error (UD)																																				
L	H	H	Bus error (BE)																																				
L	H	L	Timeout error (TO)																																				
L	L	H	Reserved																																				
L	L	L	Reserved																																				
MID3 MID2 MID1 MID0	I	AM20 AB8 AC1 AH20	MBus module ID. The identifier of this MBus device and is usually hardwired by the system. MID3 is the most significant bit (MSb) and MID0 is the least significant bit (LSb).																																				
MIF	I/O†	AP16	<p>MBus memory inhibit. This signal is asserted by a snooping cache during coherent reads when it finds it has the dirty copy of cacheable data. When MIF is asserted during a MBus transaction, memory (slave) is inhibited from responding and the snooping cache supplies the data instead. The MXCC can assert this signal when snooping MBus transactions. It senses this signal when it is either the master or a slave on an MBus transaction.</p> <p>H = Normal operation. L = Inhibit memory, snooping cache to supply data.</p>																																				
MIRL3 MIRL2 MIRL1 MIRL0	I	Y2 Y4 W5 W7	<p>MBus system interrupt request level. This field specifies the level of the highest priority interrupt request that is currently pending. If MIRL3 – MIRL0 = 0000, no interrupts are pending.</p> <p>Level 15: (MIRL3 – MIRL0 = 1111) NMI (disable all traps). Level 14: Highest maskable interrupt. Level 1: Lowest maskable interrupt. Level 0: No interrupts are pending.</p>																																				
MRDY	I/O	AH16	MBus ready. Encoded along with MERR and MRTY to indicate acknowledgment type (the type of error response). See table in MERR description.																																				
MRTY	I	AR15	MBus retry. Encoded along with MERR and MRDY to indicate acknowledgment type (the type of error response). See table in MERR description.																																				
MSH	I/O‡§	AM18	<p>MBus shared. This is asserted by snooping caches that have a valid entry matching the address of the current bus transaction.</p> <p>H = No snooping cache has a valid entry which matches the address of the current bus transaction. L = One or more snooping caches has a valid entry which matches the address of the current bus transaction.</p>																																				
OE	I/O†	E15	<p>SRAM output enable. As an output this signal controls the pipelined output enable of external cache SRAM. It is used as an input to prevent bus collisions.</p> <p>H = SRAM outputs disabled L = SRAM outputs enabled</p>																																				
PCLK	I	V34	Processor clock. Is the same clock as to the processor.																																				

† These pins have internal holding drivers.

‡ These pins have internal pullup resistors.

§ These pins have an open drain.

Table D-1. Pin Functions — MBus Configuration (MBSEL = H) (Continued)

PIN NAME	I/O	PIN NO.	FUNCTION
PEND	O	B12	Pending. A store is pending in the MXCC or on the MBus. This signal is asserted by the MXCC when it has a store operation pending internally or on the system bus. This signal indicates that at least one outstanding write operation has not completed. H = All write operations issued by this processor are completed. L = One or more write operations that were issued by this processor are not yet complete.
PLLBYP	†	AE35	PLL bypass. This pin is used to bypass both of the internal phase lock loops. When PLLBYP is asserted, PCLK directly supplies timing for the circuits in the MXCC's processor clock domain, and BCLK directly supplies timing for the circuits of the MXCC's bus clock domain. The normal delay compensation performed by the PLL is defeated. H = PLLs are enabled. Normal operation. L = PLLs are disabled. No clock delay compensation.
PPLLRC		W31	Capacitor for the phase filter of the processor clock PLL. This pin should be connected to an external capacitor to ground. With an internal resistor, this circuit provides the RC time constant for the phase filter of the processor clock domain PLL.
RD	†	A13	This signal is asserted when a read address is on ADDR35 – ADDR0. Also asserted with DEMAP to indicate completion of a bus demap operation by the processor. H = No read. L = With DEMAP: demap operation requested by the MXCC is complete. Without DEMAP: a data read request. With LDST and WR: an atomic load/store operation.
RESET	O	G25	Reset. MXCC output used to reset the processor when the system asserts RSTIN. H = Normal operation. L = Reset to processor.
RETRY	O	B14	Retry. This signal is encoded, along with RRDY or WRDY, and MEXC to indicate the type of acknowledgment. See MEXC description for table. If this signal is asserted before RRDY or WRDY is asserted for an access, the processor should terminate the current access and restart it once it reacquires the Vbus (if a processor read is pending, a processor write will not be retried until after the read has completed).
RGRT	O	H12	Read grant. This signal grants the processor read access on the VBus. H = Processor not allowed read access. L = Processor may make read accesses.
RRDY	O	A11	Read ready. This signal indicates that read data is valid. When RRDY is asserted, the processor may reliably sample the incoming data on the same clock edge as RRDY. This signal is used to qualify data specifically for a read access since a write may also be pending. This signal is encoded with MEXC and RETRY. See MEXC description for table.
RSTIN	†	AE7	Reset in. Reset from the system to the cache controller. H = Normal operation. L = Hardware reset (see reset section).
SIZE1 SIZE0	†	B26 E25	Size of data transfer. These bits indicate the transfer size of the current bus transaction initiated by the processor. SIZE1 – SIZE0 = 00 for byte transfer SIZE1 – SIZE0 = 01 for halfword transfer SIZE1 – SIZE0 = 10 for word transfer SIZE1 – SIZE0 = 11 for doubleword transfer

† These pins have internal holding drivers.

‡ These pins have internal pullup resistors.

Table D-1. Pin Functions — MBus Configuration (MBSEL = H) (Continued)

PIN NAME	I/O	PIN NO.	FUNCTION
SU	I†	G23	Supervisor access. This signal is asserted by the processor with $\overline{\text{CMD5}}$ when the access was initiated in supervisor mode. H = User (unprivileged) transaction. L = Supervisor (privileged) transaction.
SYNC	I‡	B8	Synchronous clocks. When this signal is asserted, the synchronizers are bypassed, eliminating their delay but requiring that BCLK and PCLK be identical. H = Asynchronous. PCLK and BCLK may have different rates. L = Synchronous. PCLK and BCLK must be identical.
TCK	I‡	AE5	JTAG test clock.
TDI	I‡	AF2	JTAG test data.
TDO	O	AD2	JTAG test data output or PLL output (see TEST below).
TEST	I‡	AD8	Three-state all output drivers and monitor PLL on TDO.
TMS	I‡	AE1	JTAG test mode select.
TRST	I‡	AD4	JTAG test reset.
WE0 WE1 WE2 WE3 WE4 WE5 WE6 WE7	O†	E27 B28 E29 D30 D8 G9 D6 H8	SRAM write enables. These signals directly control the write enable signals of synchronous SRAM used as external cache. These signals are driven only when asserted, otherwise they are in the high-impedance state. $\overline{\text{WE}}x$ bit ordering corresponds to the big-endian convention. That is:  <div style="display: flex; justify-content: space-between;"> <div> <math>\overline{\text{WE}}0</math>: DATA63 – DATA56  <math>\overline{\text{WE}}2</math>: DATA47 – DATA40  <math>\overline{\text{WE}}4</math>: DATA31 – DATA24  <math>\overline{\text{WE}}6</math>: DATA15 – DATA08 </div> <div> <math>\overline{\text{WE}}1</math>: DATA55 – DATA48  <math>\overline{\text{WE}}3</math>: DATA39 – DATA32  <math>\overline{\text{WE}}5</math>: DATA23 – DATA16  <math>\overline{\text{WE}}7</math>: DATA07 – DATA00 </div> </div> H = SRAM read L = SRAM write
WEE	O	H24	E-cache write enable enable. When asserted, the SSP may assert its write enables to write E-cache directly. This pin is used to control the assertion of processor's $\overline{\text{WE}}7$ – $\overline{\text{WE}}0$ signals. H = The processor may not drive $\overline{\text{WE}}7$ – $\overline{\text{WE}}0$ . L = The processor may drive $\overline{\text{WE}}7$ – $\overline{\text{WE}}0$ .
WGRT	O	E11	Write grant. This signal grants the processor write access on the VBus. H = The processor is not allowed write access. L = The processor may make write accesses.
WH	I/O†	D14	As an input, this signal is asserted with a write address on ADDR35 – ADDR0 and write data on DATA63 – DATA0. It is also asserted by the processor with $\overline{\text{DEMAP}}$ to send a demap request to the system bus. H = Not a write cycle. L = Write (or load/store with $\overline{\text{RD}}$ and $\overline{\text{LDST}}$ low or demap) cycle.  As an output the MXCC asserts this signal with an address on ADDR35 – ADDR0 to invalidate lines in the processor's internal cache(s) containing that address. H = Normal L = Demap

† These pins have internal holding drivers.

‡ These pins have internal pullup resistors.



Table D-1. Pin Functions — MBus Configuration (MBSEL = H) (Continued)

PIN NAME	I/O	PIN NO.	FUNCTION
WRDY	O	D12	Write ready. When WRDY is asserted, the MXCC has sampled the processor's write data, and so the processor may generate the next access. In the case of burst writes, the processor switches address and data for the next write within the burst on the same clock edge as WRDY was asserted. This signal is used to qualify data specifically for a write access since a read may also be pending. This signal is encoded with MEXC and RETRY. See MEXC description for table.
nu	‡	AH18	Not used in the MBus interface.
nu	‡	AP18	Not used in the MBus interface.
nu	‡	AJ19	Not used in the MBus interface.
nu	‡	AL19	Not used in the MBus interface.
nu		AR21	Not used in the MBus interface.
nu	I	AH8	Not used in the MBus interface.
nu	I	AJ23	Not used in the MBus interface.
nu	†	Y8	Not used in the MBus interface.
nu	†	AA1	Not used in the MBus interface.
nu	†	AA5	Not used in the MBus interface.
nu	†	AA7	Not used in the MBus interface.
nu		AB4	Not used in the MBus interface.
nu		AB2	Not used in the MBus interface.

† These pins have internal holding drivers.

‡ These pins have internal pullup resistors.

§ These pins have an open drain.

Table D-2. Pin Functions — XBus Configuration (MBSEL = L)

PIN NAME	I/O	PIN NO.	FUNCTION
ADDR35	I/O†	V32	Processor physical address bus.
ADDR34		Y34	
ADDR33		Y32	
ADDR32		AA35	
ADDR31		Y28	
ADDR30		AB34	
ADDR29		AA29	
ADDR28		AA31	
ADDR27		AB32	
ADDR26		AC35	
ADDR25		AD34	
ADDR24		AC31	
ADDR23		E21	
ADDR22		B24	
ADDR21		H22	
ADDR20		E23	
ADDR19		A23	
ADDR18		D22	
ADDR17		G21	
ADDR16		B22	
ADDR15		D20	
ADDR14		H20	
ADDR13		A21	
ADDR12		B20	
ADDR11		G19	
ADDR10		E19	
ADDR09		H18	
ADDR08		B18	
ADDR07		D18	
ADDR06		E17	
ADDR05		G17	
ADDR04		D16	
ADDR03		B16	
ADDR02		H16	
ADDR01		A15	
ADDR00		G15	
BCLK	I	AF34	Bus clock.
BPLLRC	I	AE31	Capacitor for the phase filter of the bus clock PLL. This pin should be connected to an external capacitor to ground. With an internal resistor, this circuit provides the RC time constant for the phase filter of the bus clock domain PLL.
BURST	I†	H14	This signal indicates whether a burst access is in progress. BURST is driven at the same time as ADDR35 – ADDR0 and it is asserted during both read bursts and write bursts. BURST is deasserted on the last address of a burst to allow the MXCC to stop returning RRDY or WRDY with the last data of the burst. H = A burst access is in progress. L = A burst access is not in progress.
CCERR	O	AC5	Indicates either an internal MXCC error or ERROR is asserted by the processor H = No error. L = An internal processor or MXCC error.

† These pins have internal holding drivers.

Table D-2. Pin Functions — XBus Configuration (MBSEL = L) (Continued)

PIN NAME	I/O	PIN NO.	FUNCTION
CCRBL	I†	D24	Cacheable access. This pin indicates the current processor transaction as one that may be cached in an external cache. H = Noncacheable access. L = Cacheable access.
CMDS	I/O†	A9	Command strobe. Indicates the beginning of a bus cycle. The VBus master asserts this signal for one cycle to begin all of its accesses. When the MXCC is a bus master, as indicated by <b>WGHT</b> and <b>RGHT</b> being deasserted, it asserts <b>CMDS</b> to initiate invalidate and demap transactions. H = Not a command word. L = VBus invalidate or demap command word on <b>ADDR35 – ADDR0</b> , <b>DEMAP</b> , and <b>WR</b> .  When the MXCC is not a bus master, this signal indicates the first cycle of a VBus transaction. H = Not a command word. L = VBus command word on <b>ADDR35–ADDR0</b> , <b>CCRBL</b> , <b>CSA</b> , <b>DEMAP</b> , <b>LDST</b> , <b>SIZE1–SIZE0</b> , <b>SU</b> , <b>RD</b> , and <b>WR</b> .
CSA	I†	G11	Control-space access. The processor asserts this signal when performing a read or write to the internal tag RAM, E-cache, or registers of the MXCC. H = Normal memory access. L = Control space access.

† These pins have internal holding drivers.

Table D-2. Pin Functions — XBus Configuration (MBSEL = L) (Continued)

DATA63		U31	
DATA62		U29	
DATA61		T34	
DATA60		T32	
DATA59		T28	
DATA58		R35	
DATA57		R29	
DATA56		P32	
DATA55		R31	
DATA54		N35	
DATA53		P28	
DATA52		M34	
DATA51		P34	
DATA50		N29	
DATA49		M32	
DATA48		L35	
DATA47		N31	
DATA46		L31	
DATA45		M28	
DATA44		L29	
DATA43	I/O†	K34	Processor data bus.
DATA42		K32	
DATA41		J35	
DATA40		H34	
DATA39		J31	
DATA38		K28	
DATA37		J29	
DATA36		H32	
DATA35		G31	
DATA34		F32	
DATA33		H28	
DATA32		D28	
DATA31		V4	
DATA30		V8	
DATA29		U5	
DATA28		U7	
DATA27		T2	
DATA26		T4	
DATA25		R1	
DATA24		T8	
DATA23		R5	

† These pins have internal holding drivers.

Table D-2. Pin Functions — XBus Configuration (MBSEL = L) (Continued)

PIN NAME	I/O	PIN NO.	FUNCTION
DATA22 DATA21 DATA20 DATA19 DATA18 DATA17 DATA16 DATA15 DATA14 DATA13 DATA12 DATA11 DATA10 DATA09 DATA08 DATA07 DATA06 DATA05 DATA04 DATA03 DATA02 DATA01 DATA00	I/O†	P2 R7 P4 N1 P8 N5 N7 L5 M2 M4 L1 L7 M8 K4 J1 K2 J5 H2 K8 H4 G5 J7 F4	Processor data bus (continued).
DEMAPP	I/O†	E13	<p>As an input, this signal is asserted with a demap data word on DATA63–DATA0 and WR asserted to pass a demap request from the processor to the MXCC, and then to the system bus. DEMAPP asserted with RD asserted indicates that the processor has successfully completed a demap operation requested by the MXCC (initiated from the system bus).</p> <p>H = No demap request. L = When WR process has requested a demap cycle. When RD process has completed a system bus requested demap cycle.</p> <p>As an output, this signal is asserted when the system bus has requested a demap operation. DATA63–DATA0 contains demap data word indicating which virtual address translations are to be discarded.</p> <p>H = No demap request. L = System bus requested demap cycle.</p>
DPAR0 DPAR1 DPAR2 DPAR3 DPAR4 DPAR5 DPAR6 DPAR7	I/O†	D26 A27 H26 G27 D10 E9 E7 H10	<p>Data bus parity. When parity is enabled, even parity is generated and checked. DPAR0 is parity for bits DATA63–DATA56. When parity checking is disabled, odd parity is generated but not checked.</p> <p>DPAR0: DATA63 – DATA56      DPAR1: DATA55 – DATA48 DPAR2: DATA47 – DATA40      DPAR3: DATA39 – DATA32 DPAR4: DATA31 – DATA24      DPAR5: DATA23 – DATA16 DPAR6: DATA15 – DATA08      DPAR7: DATA07 – DATA00</p>
ERROR	I†	A25	<p>Processor error. The processor asserts this pin when it has entered an internal error state. The MXCC initiates an internal reset when ERROR is asserted.</p> <p>H = Normal operation. L = Processor internal error.</p>

† These pins have internal holding drivers.

Table D-2. Pin Functions — XBus Configuration (MBSEL = L) (Continued)

PIN NAME	I/O	PIN NO.	FUNCTION																																													
GTLREF	I	AH8	XBus level reference for GTL and GTL/TTL selection. Should be connected to a voltage source of $V_{ref}$ for GTL operation of the XBus interface signals. Should be connected to $V_{CC}$ for TTL operation of the XBus interface signals. Since this pin (and GTLREF1) sets threshold levels, care should be taken to insure that $V_{ref}$ is free of noise. GTLREF and GTLREF1 are connected together internally.																																													
GTLREF1	I	AJ23	XBus level reference for GTL and GTL/TTL selection. GTLREF and GTLREF1 are connected together internally.																																													
IRL3 IRL2 IRL1 IRL0	O	AB28 AC29 AD32 AE29	Interrupt request Level. This field specifies, to the processor, the level of the highest priority interrupt request that is currently pending. If IRL3 - IRL0 = 0000, no interrupts are pending. Level 15: (IRL3 - IRL0 = 1111): Nonmaskable interrupt. Level 14: Highest maskable interrupt. Level 1: Lowest maskable interrupt. Level 0: No interrupts are pending.																																													
LCMD2 LCMD1 LCMD0	O	AB4 AC1 AB8	Boot-bus command bits. Commands are issued by the MXCC and interpreted by one or more external Boot Bus controllers. <table><thead><tr><th>LCMD2</th><th>LCMD1</th><th>LCMD0</th><th>NAME</th><th>MEANING</th></tr></thead><tbody><tr><td>H</td><td>H</td><td>H</td><td>ADR-HIGH</td><td>Address bits 23 - 16 on LDATA</td></tr><tr><td>H</td><td>H</td><td>L</td><td>Interrupt</td><td>Interrupt Status on LDATA</td></tr><tr><td>H</td><td>L</td><td>H</td><td>ADR-MED</td><td>Address bits 15 - 8 on LDATA</td></tr><tr><td>H</td><td>L</td><td>L</td><td>ADR-LOW</td><td>Address bits 7 - 0 on LDATA</td></tr><tr><td>L</td><td>H</td><td>H</td><td>IDLE-WR</td><td>Idle for write</td></tr><tr><td>L</td><td>H</td><td>L</td><td>READ-VALID</td><td>Device data on LDATA</td></tr><tr><td>L</td><td>L</td><td>H</td><td>WRITE-VALID</td><td>MXCC data on LDATA</td></tr><tr><td>L</td><td>L</td><td>L</td><td>IDLE</td><td>Idle</td></tr></tbody></table>	LCMD2	LCMD1	LCMD0	NAME	MEANING	H	H	H	ADR-HIGH	Address bits 23 - 16 on LDATA	H	H	L	Interrupt	Interrupt Status on LDATA	H	L	H	ADR-MED	Address bits 15 - 8 on LDATA	H	L	L	ADR-LOW	Address bits 7 - 0 on LDATA	L	H	H	IDLE-WR	Idle for write	L	H	L	READ-VALID	Device data on LDATA	L	L	H	WRITE-VALID	MXCC data on LDATA	L	L	L	IDLE	Idle
LCMD2	LCMD1	LCMD0	NAME	MEANING																																												
H	H	H	ADR-HIGH	Address bits 23 - 16 on LDATA																																												
H	H	L	Interrupt	Interrupt Status on LDATA																																												
H	L	H	ADR-MED	Address bits 15 - 8 on LDATA																																												
H	L	L	ADR-LOW	Address bits 7 - 0 on LDATA																																												
L	H	H	IDLE-WR	Idle for write																																												
L	H	L	READ-VALID	Device data on LDATA																																												
L	L	H	WRITE-VALID	MXCC data on LDATA																																												
L	L	L	IDLE	Idle																																												
LCMDS	O	AB2	Boot bus command strobe. When asserted, this signal indicates that command information on LCMD (and write data on LDATA for WRITE-VALID commands) is valid. Input data is latched on the rising edge. H = Inactive. L = Bus command valid.																																													
LDATA7 LDATA6 LDATA5 LDATA4 LDATA3 LDATA2 LDATA1 LDATA0	I/O	W5 W7 Y2 Y4 Y8 AA1 AA5 AA7	Boot-bus address/data.																																													
LDST	†	B10	This signal indicates an atomic load/store (LDSTUB, LDSTUBA, SWAP, or SWAPA) operation. It is equivalent to the logical OR of RD and WR signals. No other transactions may occur while LDST is asserted. H = No LDST. L = Atomic load/store (LDST) cycle.																																													
MBSEL	‡	V2	MBus select. This signal is used to select the system bus interface. This signal should not be changed during operation of this device. H = MBus system interface. L = XBus system interface.																																													

† These pins have internal holding drivers.

‡ These pins have internal pullup resistors.

Table D-2. Pin Functions — XBus Configuration (MBSEL = L) (Continued)

PIN NAME	I/O	PIN NO.	FUNCTION
MEXC	O	G13	Memory exception. This signal is asserted when the memory controller could not return or accept the requested data. This signal may cause the processor to take a memory exception trap. This signal is encoded with <b>RRDY</b> or <b>WRDY</b> , and <b>RETRY</b> to indicate the type of acknowledgment.
			<b>MEXC</b> <b>RRDY/WRDY</b> <b>RETRY</b> <b>Description</b>
			H              H              H              No reply
			H              H              L              Retry
			H              L              H              Data transfer complete
			H              L              L              Undefined error (UD)
			L              H              H              Bus error (BE)
			L              H              L              Timeout error (TO)
OE	I/O†	E15	SRAM output enable. As an output, this signal controls the pipelined output enable of external cache SRAM. It is used as an input to prevent bus collisions.
			H = SRAM outputs disabled. L = SRAM outputs enabled.
PCLK	I	V34	Processor clock. Should be the same as clock to the processor.
PEND	O	B12	Pending. A store is pending in the MXCC or in the system beyond the MXCC. This signal is asserted by the MXCC when it has a store operation pending internally or on the system bus. This signal indicates that at least one outstanding write operation has not completed. H = No incomplete write operations outstanding from this processor. L = One or more write operations issued by this processor are not yet complete.
PLLBYP	†‡	AE35	PLL bypass. This pin is used to bypass both of the internal phase lock loop. When <b>PLLBYP</b> is asserted <b>PCLK</b> directly supplies timing for the circuits in the MXCC's processor clock domain, and <b>BCLK</b> directly supplies timing for the circuits of the MXCC's bus clock domain. The normal delay compensation performed by the PLL is defeated. H = PLLs are enabled. Normal operation. L = PLLs are disabled. No clock delay compensation.
PPLSRC		W31	Capacitor for the phase filter of the processor clock PLL. This pin should be connected to an external capacitor to ground. With an internal resistor, this circuit provides the RC time constant for the phase filter of the processor clock domain PLL.
RD	†	A13	This signal is asserted when a read address is on <b>ADDR35 – ADDR0</b> . Also asserted with <b>DEMAP</b> to indicate completion of a bus demap operation by the processor. H = No read. L = With <b>DEMAP</b> : demap operation requested by the MXCC is complete. Without <b>DEMAP</b> : a data read request. With <b>LDST</b> and <b>WR</b> : an atomic load/store operation.
RESET	O	G25	Reset. This MXCC output is used to reset the processor when the system asserts <b>RSTIN</b> . H = Normal operation. L = Reset to processor.
RETRY	O	B14	Retry. This signal is encoded, along with <b>RRDY</b> or <b>WRDY</b> , and <b>MEXC</b> to indicate the type of acknowledgment. See <b>MEXC</b> description for table. If this signal is asserted before <b>RRDY</b> or <b>WRDY</b> is asserted for an access, the processor should terminate the current access and restart it once it reacquires the Vbus (if a processor read is pending, a processor write will not be retried until after the read has completed)
RGRT	O	H12	Read grant. This signal grants the processor read access on the VBus. H = Processor not allowed read access. L = Processor may make read accesses.

† These pins have internal holding drivers.

‡ These pins have internal pullup resistors.

Table D-2. Pin Functions — XBus Configuration (MBSEL = L) (Continued)

PIN NAME	I/O	PIN NO.	FUNCTION
RRDY	O	A11	Read ready. This signal indicates that read data is valid. When RRDY is asserted, the processor may reliably sample the incoming data on the same clock edge as RRDY. This signal is used to qualify data specifically for a read access since a write may also be pending. This signal is encoded with MEXC and RETRY. See MEXC description for table.
RSTIN	†	AE7	Reset in. Reset from the system to the cache controller. H = Normal operation. L = Hardware reset (see reset section).
SIZE1 SIZE0	†	B26 E25	Size of data transfer. These bits indicate the transfer size of the current bus transaction initiated by the processor. SIZE1 - SIZE0 = 00 for byte transfer SIZE1 - SIZE0 = 01 for halfword transfer SIZE1 - SIZE0 = 10 for word transfer SIZE1 - SIZE0 = 11 for doubleword transfer
SU	†	G23	Supervisor access. This signal is asserted by the processor with CMDS when the access was initiated in supervisor mode. H = User (unprivileged) transaction. L = Supervisor (privileged) transaction.
SYNC	†	B8	Synchronous clocks. When this signal is asserted, the synchronizers are bypassed, eliminating their delay, but requiring that BCLK and PCLK be identical. H = Asynchronous. PCLK and BCLK may have different rates. L = Synchronous. PCLK and BCLK must be identical.
TCK	†	AE5	JTAG test clock.
TDI	†	AF2	JTAG test data.
TDO	O	AD2	JTAG test data output or PLL output (see TEST below).
TEST	†	AD8	3-state all output drivers and monitor PLL on TDO.
TMS	†	AE1	JTAG test mode select.
TRST	†	AD4	JTAG test reset.
WE0 WE1 WE2 WE3 WE4 WE5 WE6 WE7	O†	E27 B28 E29 D30 D8 G9 D6 H8	SRAM write enables. These signals directly control the write enable signals of synchronous SRAM used as external cache. These signals are driven only when asserted, otherwise they are in the high-impedance state. WEx bit ordering corresponds to the big-endian convention. That is:  <div style="display: flex; justify-content: space-between;"> <div> WE0: DATA63 - DATA56 WE2: DATA47 - DATA40 WE4: DATA31 - DATA24 WE6: DATA15 - DATA08 </div> <div> WE1: DATA55 - DATA48 WE3: DATA39 - DATA32 WE5: DATA23 - DATA16 WE7: DATA07 - DATA00 </div> </div> H = SRAM read L = SRAM write
WGRT	O	E11	Write grant. This signal grants the processor write access on the VBus. H = The processor not allowed write access. L = The processor may make write accesses.

† These pins have internal holding drivers.

‡ These pins have internal pullup resistors.



Table D-2. Pin Functions — XBus Configuration (MBSEL = L) (Continued)

PIN NAME	I/O	PIN NO.	FUNCTION
WR	I/O†	D14	<p>As an input, this signal is asserted with a write address on ADDR35 – ADDR0 and write data on DATA63 – DATA0. It is also asserted by the processor with DEMAP to send a demap request to the system bus.</p> <p>H = Not a write cycle. L = Write (or load/store with RD and CDST low or demap) cycle.</p> <p>As an output, the MXCC asserts this signal with an address on ADDR35 – ADDR0 to invalidate lines in the processor's internal cache(s) containing that address.</p> <p>H = Normal. L = Demap.</p>
WRDY	O	D12	<p>Write ready. When WRDY is asserted, the MXCC has sampled the processor's write data, and so the processor may generate the next access. In the case of burst writes, the processor switches address and data for the next write within the burst on the same clock edge as WRDY was asserted. This signal is used to qualify data specifically for a write access since a read may also be pending. This signal is encoded with MEXC and RETRY. See MEXC description for table.</p>

† These pins have internal holding drivers.

Table D-2. Pin Functions — XBus Configuration (MBSEL = L) (Continued)

XDATA63		AF32	
XDATA62		AG35	
XDATA61		AG31	
XDATA60		AH34	
XDATA59		AH32	
XDATA58		AG29	
XDATA57		AJ31	
XDATA56		AK32	
XDATA55		AH28	
XDATA54		AM30	
XDATA53		AL29	
XDATA52		AM28	
XDATA51		AJ27	
XDATA50		AP28	
XDATA49		AH26	
XDATA48		AL27	
XDATA47		AR27	
XDATA46		AM26	
XDATA45		AP26	
XDATA44		AJ25	
XDATA43		AH24	
XDATA42		AR25	
XDATA41		AM24	
XDATA40		AP24	
XDATA39		AL25	
XDATA38		AL23	
XDATA37	I/O <sup>§</sup>	AH22	XBus multiplexed command / data bus.
XDATA36		AR23	
XDATA35		AM22	
XDATA34		AP22	
XDATA33		AL21	
XDATA32		AJ21	
XDATA31		AJ15	
XDATA30		AP14	
XDATA29		AM14	
XDATA28		AR13	
XDATA27		AL13	
XDATA26		AH14	
XDATA25		AP12	
XDATA24		AJ13	
XDATA23		AM12	
XDATA22		AR11	
XDATA21		AL11	
XDATA20		AP10	
XDATA19		AH12	
XDATA18		AM10	
XDATA17		AJ11	
XDATA16		AR9	
XDATA15		AP8	
XDATA14		AM8	
XDATA13		AH10	
XDATA12		AJ9	
XDATA11		AL9	
XDATA10		AL7	

<sup>§</sup> These pins have an open drain.

Table D-2. Pin Functions — XBus Configuration (MBSEL = L) (Continued)

PIN NAME	I/O	PIN NO.	FUNCTION
XDATA9 XDATA8 XDATA7 XDATA6 XDATA5 XDATA4 XDATA3 XDATA2 XDATA1 XDATA0	I/O <sup>§</sup>	AM6 AK4 AJ5 AG7 AF8 AG5 AH4 AH2 AG1 AF4	XBus multiplexed command / data bus (continued).
XREQ0[1] XREQ0[0]	I <sup>‡</sup>	AJ17 AM16	Request field from Bus Watcher 0 (BW0).
XREQ1[1] XREQ1[0]	I <sup>‡</sup>	AM18 AP16	Request field from Bus Watcher 1 (BW1).
XREQ2[1] XREQ2[0]	I <sup>‡</sup>	AH18 AP18	Request field from Bus Watcher 2 (BW2).
XREQ3[1] XREQ3[0]	I <sup>‡</sup>	AJ19 AL19	Request field from Bus Watcher 3 (BW3).
XPAR3 XPAR2 XPAR1 XPAR0	I/O <sup>§</sup>	AL17 AR15 AH16 AL15	Parity bits. XPAR3 = Parity over XDATA63 – XDATA48,    XPAR2 = Parity over XDATA47 – XDATA32 XPAR1 = Parity over XDATA31 – XDATA16,    XPAR0 = Parity over XDATA15 – XDATA0
XGNT0	O <sup>§</sup>	AP20	XBus Grant to Bus Watcher 0 (BW0).
XGNT1	O <sup>§</sup>	AM20	XBus Grant to Bus Watcher 1 (BW1).
XGNT2	O <sup>§</sup>	AH20	XBus Grant to Bus Watcher 2 (BW2).
XGNT3	O <sup>§</sup>	AR21	XBus Grant to Bus Watcher 3 (BW3).

<sup>‡</sup> These pins have internal pullup resistors.<sup>§</sup> In GTL operation, the I/O buffer is open-drain, while in TTL operation the I/O buffer is 3-state.

Table D-3. Pin Functions — Power Connections

PIN NAME	I/O	PIN NO.	FUNCTION
V <sub>CCC</sub>	I	C15, C21, F10, F26, K6, K30, R3, R33, AA3, AA33, AF6, AF30, AK10, AK26, AN15, AN21	Supply voltage (V <sub>CC</sub> ) for internal (core) logic.
V <sub>CCCKB</sub>	I	AF28	Supply voltage (V <sub>CC</sub> ) for bus clock and PLL.
V <sub>CCCKP</sub>	I	W29	Supply voltage (V <sub>CC</sub> ) for processor clock and PLL.
V <sub>CCPX</sub>	I	Y6, AG3, AG33, AK16, AK20, AL5, AL31, AN9, AR17, AN27	Supply voltage (V <sub>CC</sub> ) for bus outputs.
V <sub>CCI</sub>	I	C23, G7, N33, AC3, AJ29, AN13	Supply voltage (V <sub>CC</sub> ) for inputs.
V <sub>CCP</sub>	I	A19, C9, C27, E5, E31, F16, F20, J3, J33, T6, T30, U35, W1, Y31	Supply voltage (V <sub>CC</sub> ) for processor outputs.
V <sub>SSC</sub>	I	C17, C19, F8, F12, F24, F28, H6, H30, M6, M30, U3, U33, W3, W33, AD6, AD30, AH6, AH30, AK8, AK12, AK24, AK28, AN17, AN19	Ground for internal (core) logic.
V <sub>SSCKB</sub>	I	AD28	Ground for bus clock and PLL.
V <sub>SSCKP</sub>	I	V28	Ground for processor clock and PLL.
V <sub>SSI</sub>	I	C13, G29, N3, AC33, AJ7, AN23	Ground for inputs.
V <sub>SSP</sub>	I	A17, C7, C11, C25, C29, F6, F14, F18, F22, F30, G3, G33, L3, L33, P6, P30, U1, V6, V30, W35, AB30	Ground for processor outputs.
V <sub>SSPX</sub>	I	AB6, AE3, AE33, AJ3, AJ33, AK6, AK14, AK18, AK22, AK30, AN7, AN11, AN29, AR19, AN25	Ground for bus outputs.



## **SuperSPARC Revision Summary**

---

---

---

This appendix contains a summary of the different SuperSPARC microprocessor revisions in tabular form.



## SuperSPARC Revision Differences Summary

FEATURE	PART NUMBER			COMMENTS
	STP1020NPGA	STP1020PGA	STP1020APGA	
Applications	Uniprocessor systems in Stand-alone mode. Multiprocessor systems when used with MXCC.	Uniprocessor or Multiprocessor systems.	Uniprocessor or Multiprocessor systems.	
Maximum Clock Frequency	40 MHz	50 MHz	60 MHz	STP1020A is also available in 50 MHz.
Maximum VBus Frequency	40 MHz	50 MHz	60 MHz	
Maximum MBus Frequency	40 MHz	50 MHz	50 MHz	
Maximum Power Dissipation	9.0 Watts	12.8 Watts	15.4 Watts (Approx, with Icc = 2.8A)	Max Power = Icc@Vcc (Max) * Vcc (Max)
Normal Voltage	5.3V	5.3V	5.0V	
Tolerance of Voltage	+/- 50mV	+/- 50 mV	+/- 50mV	For all parts, a regulator (module) is used to maintain the Vcc within the tolerance.
Max Junction Temperature	90C	90C	85C	
Functionality of Pin AH30 in VBus mode	spare3; Not Used; externally pulled High or left floating.	spare3; Not Used; externally pulled High or left floating	ADDR20*; Driven out by the CPU	Inverted ADDR20 signal to support 2MB ext cache.
Functionality of Pin AH30 in MBus mode	spare3; Not Used; externally pulled High or left floating	spare3; Not Used; externally pulled High or left floating	spare3; Not Used; externally pulled High or left floating	
Processor State Register PSR (mp bits[31:28])	0x4	0x4	0x4	This field is read with the privileged instruction RDPSPR.
Processor State Register PSR (ver bits[7:24])	0x1	0x0	0x0	This field is read with the privileged instruction RDPSPR.
MMU Control Register MCNT (mp bits[31:28] value)	0x0	0x0	0x0	Accessed with ASI 0x04, and VAI(2:0)=0x0
MMU Control Register MCNT (ver bits[27:24] value)	0x0	0x4	0x3	Accessed with ASI 0x04, and VAI(2:0)=0x0.
BIST Short Signature	0x1EAA70AC	0x488F2256	To be determined	Accessed with ASI 0x39, or obtained with JTAG scan.
JTAG Component ID	0x0000402F	0x1000402F	0x3000402F	Obtained with JTAG scan.

Last Revised 4/12/94





## **MultiCache Controller Revision Summary**

---

---

This appendix contains a summary of the MultiCache Controller revisions in tabular form.



## MXCC Revision Differences Summary

FEATURE	PART NUMBER		COMMENTS
	STP1090	STP1090A	
Applications	Uniprocessor or Multiprocessor systems.	Uniprocessor or Multiprocessor systems.	
Maximum Clock Frequency	60 MHz	60 MHz	
Maximum VBus Frequency	60 MHz	60 MHz	
Maximum MBus Frequency	50 MHz	50 MHz	
Maximum XBus Frequency	50 MHz	50 MHz	
Nominal Voltage	5 V	6 V	
Range of Voltage	4.75 V - 5.25 V	4.75 V - 5.25 V	
Maximum Power Dissipation	6.3 Watts	6.3 Watts	Max Power = $I_{cc@V_{cc}(\text{Max})} \cdot V_{cc}(\text{Max})$
Max Junction Temperature	80 °C	80 °C	
Functionality of AC7 Pin	Spare	Inverted ADDR20 driven out by MXCC.	Inverted ADDR20 signal to support 2MB ext cache . Eliminates external inverter.
Cache Invalidates on First Stores ( or writes to clean lines ) in MBus	17 Cycles	5 Cycles	
Non-Cacheable Store Latency in MBus	17 Cycles	6 Cycles	
Non-Cacheable Stores in XBus	Single Store in TSO mode	Multiple Stores in TSO mode till Cacheable Store occurs	
Writeback Performance during sub-block flushing in MBus, XBus	One sub-block flushed at a time	1,2, or sub-blocks flushed at once	
Snoop Data Latency in MBus	A+18	A+12 ( 6 Cycles Improvement)	
VBus arbitration		Improved Latencies for Load Misses and Prefetches by 1 - 2 cycles	
MBus Port Register mrev bits[7..4] value	0x4	0xB	
BIST Signature XBus GTL	0x02B8CBB4	0x2A272030	
BIST Signature XBus TTL	0x02B8CBB4	0x2A272030	
BIST Signature MBus	0x02B8CBB4	0x2A272030	
JTAG Component ID	0x4000302F	0x8000302F	

Last Revised 4/11/94



# Glossary

---

## A

**abort:** To terminate the execution of an instruction once it has begun but before it has completed. Aborted instructions do not make any program visible changes to the state of the processor registers or to memory. An aborted instruction does not change any condition codes and cannot cause a trap. Instructions are aborted if an earlier instruction causes a trap or if the instruction was started speculatively under assumptions that proved false, such as an incorrectly predicted branch.

**ALU:** Arithmetic Logical Unit. The logic block that computes a result from one or two operands for any arithmetic or logical operation. The SuperSPARC processor contains three ALUs to process ALUops.

**ALUop:** An ALU operation. ALU operations are any instruction that computes integer arithmetic results, such as ADD, SUBcc, MULSc. Instructions that compute logical results are also ALUops; examples include AND, ORcc, SETHI and shift instructions.

**anti-dependency:** A condition that occurs when the output of one instruction overwrites an input of an earlier instruction which still requires its input. Instructions which might yet trap and be retried require their inputs to remain unaltered. See also **dependency**.

**arbiter:** That which controls access to a shared resource among several potential users. Generally, an entity that requires the resource must request the right to use it from the arbiter. The arbiter grants access to requesters, choosing between them in some way such as priority order or fair allocation to all requesters. Arbiters are most often associated with multiple-master busses.

**ASI:** Address Space Indicator. An ASI is logically appended to a processor generated logical address before it is sent to the MMU. The MMU interprets the ASI value to control mapping and protection. The ASI can also select special address spaces for diagnostic and control access to structures internal to the processor (e.g., cache tags) or external to the processor (e.g., external cache control).

**atomic operations:** Operations that perform a memory read and a memory write without allowing other access to the addressed location between the read and write portions of the operation. The SWAP and LDSTUB instructions perform atomic operations.

## B

**bandwidth:** The capacity of a data transmission medium expressed information per time. Bandwidth is most often used here to describe the data carrying capacity of busses. The units of bandwidth are usually megabytes per second. For example, a 64-bit bus running a 50 MHz has a bandwidth of at most 400 MB/s. After overhead for arbitration, address cycles, etc. are subtracted, the available bandwidth might be only 250 MB/s.

**BIST:** Built-In Self-Test. Any internal mechanism that can perform testing without an external test controller. The SuperSPARC processor and MultiCache Controller have scan-based BIST. Scan-based BIST uses a pseudo-random pattern generator to provide patterns to the internal scan paths. The patterns stimulate logic on the chip; the results of each pattern's action on the logic are captured in registers and scanned into a signature analyzer. The BIST controller performs many generate, scan, stimulate, capture, and analyze cycles, and the signatures accumulate in the signature analyzer. When the BIST cycles are complete, the signature register contains a pattern that indicates probably good logic if the pattern matches a known good pattern for the device and revision.

**boot mode:** A special MMU bypass mode in which all instruction fetches and accesses through the instruction access ASIs (0x08 and 0x09) generate the physical address by passing virtual address bits 27 through 0 unaltered and setting the upper eight bits (bits 35 through 28) of the physical address to 1s (0xFF). Hardware reset enables this translation mode.

**branch target queue:** A FIFO buffer that holds instructions to be issued if the conditional DCTI in progress chooses to transfer control. By prefetching the instructions at the target of the CTI before it is known if they will be needed, the SuperSPARC processor can execute control transfer instructions more quickly.

**breakpoint:** A point to interrupt the execution of a program for debugging purposes. A breakpoint can be raised by an instruction (such as SIGM) placed into the program or by access to an address within the range of address breakpoint comparators.

**BROP:** A branch operation. The branch operations are: BRicc, FBfcc, JMPL, CALL, and RETT instructions.

**bubble:** An instruction group containing no instructions. A bubble is most often the result of an empty instruction queue. The empty group proceeds through the pipeline stages like any other group, but contains no instructions. A bubble cannot cause a trap, not even for an interrupt.

**bus keeper:** A buffer whose input is connected to the bus and whose output connected to the bus via a relatively large resistor or other means to limit its output current. This circuit keeps the bus at the last driven value when it is not driven. Bus keepers prevent the bus signals from drifting to voltages near the threshold, which could generate noise on chips.

**bus master:** The unit that initiates a transaction on a bus. The master sends a request or command to the bus slave.

**bus slave:** The unit addressed by the bus master in a transaction on a bus. The slave acts on the request or command and responds with data or an acknowledgement.

**bus watcher:** A device or unit that interfaces between XBus and a system bus. A bus watcher (BW) must convert bus protocols as needed, manage communications over XBus with the MXCC, and manage communications over the system bus with main memory, peripheral devices, and other processors. A BW must keep a duplicate set of E-cache tags, snoop transactions on the system bus, and notify the MXCC of any changes in cache sub-block's state based on the system bus's consistency algorithm.



**C**

**cache:** A small, fast memory close to the processor that can act as a surrogate for some words of main memory. The cache establishes temporary associations between words in the cache memory and main memory addresses. On a memory reference, the processor checks the cache memory for such an association, and, if there is one, the copy in cache memory is used instead of the one in main memory. Cache memories are usually managed automatically, with words being transferred from main memory into the cache memory without any program intervention.

**cacheability:** A property of each memory access that determines whether the data will be stored in caches and should therefore interact with other caches to maintain cache consistency. Cacheability can be determined in several ways, but normal accesses when the MMU is enabled determine cacheability as a property of the virtual page table entry (PTE).

**cache block:** Cached data sharing a single address tag. Block sizes in a SuperSPARC system vary between 64 and 128 bytes, depending on the cache and the configuration. A block may have several sub-blocks, each of which may be separately valid or invalid.

**cache consistency:** The state in which all caches and main memory have the same values for all valid copies of data. Cache memories function by making copies of data. The copies might become inconsistent with main memory or with each other, which could lead to unexpected or incorrect results.

**cache line:** A term some authors prefer to **cache block**. Cache line and cache block are synonymous. This User's Guide uses "cache block."

**cache set:** The group of cache blocks from which to choose when accessing data from some single address. If there is only one block in which the data for the address may be found, the cache is called "direct mapped". If the data for the address might be found in any block in the cache, the cache is called "fully associative". If the data might be found in one of a small number ( $n$ ) of blocks in the cache, the cache is called " $n$ -way set associative". Each address has a set of  $n$  cache locations with which it might be associated. The number of blocks in the cache is the product of the number of sets times the associativity.

**cache sub-block:** If a cache block has more than one set of data storage locations with separate valid (or invalid) bits, each set of locations with its own valid bit is a sub-block. Sub-blocks allow the amount of data transferred to or from memory to be smaller than the amount of cached data controlled by a single address tag.

**cache tag:** The address and other information associated with a word or group of words that form an entry in a cache memory. The address indicates the main memory address of the word(s) in the cache entry. Other tag information includes an indicator for valid or invalid entries.

**circuit-switched bus:** A circuit-switched bus is busy from the beginning of a transaction until the transaction is complete. So, when a master uses the bus to request data from a slave, the bus is unavailable for other use until the slave delivers the data or an error. A circuit-switched bus, therefore, delivers a lower bandwidth than a **packet-switched bus** in similar technology but does not require complicated interface controllers.

**clean:** A term applied to a cache block that has not been altered since it was read from memory. See also **dirty** (clean is the opposite of dirty).

**context:** A single hardware-supported address space. The MMU supports contexts as a way to map the separate address spaces and protections of different processes onto the memory provided in hardware. A context normally corresponds one-to-one with software processes.

**copy-back:** Writing the contents of a dirty cache block or sub-block to memory. Copy-back is initiated when a dirty cache block is replaced or flushed.

**CPI:** Cycles Per Instruction. (Usually the average number of cycles per instruction.) Lower numbers yield higher performance. CPI is determined by dividing the total number of cycles to run a program by the total number of instructions executed by the program.

**CTI:** Control Transfer Instruction. Any instruction that alters the normal sequential execution of instructions. The SPARC CTIs are: Bicc, FBtcc, CALL, JMPL, RETT and Ticc instructions. Most SPARC CTIs are actually DCTIs.

**CWP:** Current Window Pointer. This register is defined in the *SPARC Architecture Manual*. The CWP indicates which of the register windows supported in a processor is currently accessed. A register specifier (either a source or a destination register specifier) in an instruction selects one of the 24 registers in a register window or one of the 8 global registers.

**cycle:** Clock cycle. The period from one active edge of the clock signal to the next occurrence of the same edge. A cycle is the basic unit of timing processor operations, as most internal operations are performed in integral numbers of cycles. See also **phase**.

**D**

**data forwarding:** Sending the result of one instruction directly to the execution of another instruction without first storing it in a register. Data forwarding saves one cycle in cases where the second instruction must wait for the result of the first. This is particularly useful in a pipelined processor.

**DCTI:** Delayed control transfer instruction. A DCTI changes execution to a new non-sequential location, after first executing one instruction sequentially after the DCTI. Most SPARC CTI instructions are DCTIs. All Bicc and FBfcc instructions, except for BA,a and FBA,a, are delayed. CALL and JMWL instructions are also delayed. Ticc is not delayed.

**demand fetch:** A block read into the instruction cache performed because an instruction in the block is needed immediately for execution. The processor cannot continue until the demand fetch is satisfied.

**demand miss:** A block read from memory into the data cache performed because a load instruction failed to find an association in the data cache for the address being accessed. The processor will wait on the memory read before continuing.

**demap:** An operation that removes one or more address translations from the TLB. In XBus configurations, demap operations are also propagated to and received from the system bus.

**denormalized:** A floating point number with the smallest exponent of its floating point format. Since the exponent is already the smallest value available, the exponent cannot be adjusted to the normal format where the value in the fraction has its most significant 1 in the bit position just beyond the most significant bit of the fraction. Since every normalized value has a 1 in this position, it is omitted from the normalized number since its value of 1 is implied.

**dependency:** The requirement to use a value or resource from an earlier instruction. A value dependency is the use of a register source computed in an earlier instruction. If the value has not yet been computed, the later instruction must wait for the dependent data. Similarly, an instruction may require access to a resource being used by another instruction and must wait for the earlier instruction to finish. See also anti-dependency.

**dirty:** A cache block is dirty if it has been altered in the cache by a store. Dirty is the opposite of **clean**. Dirty cache blocks need to be copied back to memory if the block is replaced or flushed.

**dynamic grouping:** Selecting the instructions to run simultaneously as a group, by examining the next few instructions available. This can be contrasted against static grouping, in which instruction groups are marked by the compiler or programmer.

## E

**Ecache:** External Cache. An external cache memory supplements internal instruction and data caches and provides a much larger cache using several external SRAMs.

**endian:** The way in which the constituent parts of a whole data structure are organized. The big endian organization places the first byte of a word in the highest order part of the word, while the little endian organization places the first byte of a word in the lowest order part of the word. The SPARC architecture version 8 is a big endian architecture.

**error:** A hardware-detected fault that requires software intervention. An error is frequently not recoverable and may require that the selected process be terminated. An example of an error is a bus timeout.

**error mode:** A SPARC processor enters error mode if it encounters a trapping exception or error while the PSR's enable traps (ET) flag is off. In error mode, the processor responds only to reset.

**exception:** A condition that requires software intervention. An exception is frequently recoverable to allow the program to continue after a handler has intervened. An example of an exception is when the MMU detects a missing page translation for a page that is on disk rather than in memory (page fault).

**exclusive:** A cache block is in an exclusive state if no other cache in the system has a copy of it. Exclusive is the opposite of **shared**.

## F

**fault:** A hardware or software failure that may be manifested as an error. The fault is what is broken (for example an open wire or a shorted transistor) while the error is the incorrect operation or result caused by the fault. See **error**.

**fcc:** Floating point condition code. A two-bit field in the floating-point state register (FSR) which encodes the results of floating-point compare (FCMP) instructions. Floating-point branch on floating-point condition code (FBfcc) instructions test the fcc field to decide whether to take a conditional branch. The two bits encode equal, greater, less, or unordered.

**flush:** To remove from a cache in such a way as to cause dirty data to be copied back to memory. SuperSPARC has no direct way for software to invoke a flush of the data cache.

**forwarding:** See **data forwarding**.

**FPEv:** Floating Point Event. FPEvs are integer instructions that interact with the FPU. Loads into floating-point registers, stores from floating-point registers, and branches based on floating-point condition codes are all FPEvs.

**FPop:** Floating-Point Operation. An operation that takes operands only from the floating-point registers and places results in the floating-point registers. FADDS (Floating-Point Add Single) is an example of an FPop.

**FPU:** Floating Point Unit. The part of the processor that computes floating-point results. It also contains the floating-point registers. The FPU executes all **FPOps** and participates with the IU in **FPEvs**.

## G

**global register:** Integer registers accessible from any register window. Global registers are the only general purpose integer registers in a SPARC processor that are not affected by changing the CWP.

**group:** See **instruction group**.

**GTL:** Gunning Transceiver Logic. A high-performance, low-voltage signaling technology used on MXCC in XBus configurations.

## H

**Harvard architecture:** An architecture named after the Harvard Mark-III and Mark-IV computers which had separate memories for data and instructions. Today the term is most commonly used to describe computers with separate instruction and data caches, even though those caches store information from a single memory address space. That is how this term is used in this document.

**hit:** That which occurs when an access attempt to a cache memory finds an association for the address of the access. The data can be accessed immediately in the cache.

**hold:** A pipeline hold is the same as a **stall**.

## I

- icc:** Integer condition codes. A four-bit field in the PSR that contains the integer condition code bits. The icc bits are updated by ALUop instructions whose names end in "cc" (for example, ADDcc). ALUops without the "cc" suffix and other types of instructions do not modify the icc. The branch on integer condition codes (BRicc) instructions test the condition codes to decide whether to take a conditional branch. Extended arithmetic instructions (for example ADDXcc) use the carry condition code as an input. The four condition code bits are C (carry), Z (zero), N (negative) and V (overflow).
- In registers:** The in registers of a register window are the same as the out registers of the caller's window. The in registers can be used to receive arguments from the caller and to return results to the caller.
- In-order completion:** A term applied to instructions that complete in the order in which they were issued.
- Instruction group:** A set of instructions from a program that are issued together in a superscalar processor and which execute simultaneously.
- Instruction issue:** A step in the execution of an instruction where its execution is begun. The point of issue is usually beyond instruction fetch and at the point where the instruction is decoded. An instruction must be issued to be executed; however, since some instructions may be aborted due to traps, not every instruction that is issued will complete execution.
- Interrupt:** A notice of an event, usually asynchronous and external to the processor, that requires attention of the processor.
- Invalidate:** An operation that removes an entry from a cache by marking the entry invalid. No data is transferred by an invalidate operation. An invalidated entry is no longer accessible and is available for allocation to a new entry.
- IPC:** Instructions per Cycle. The inverse of CPI. Higher numbers indicate higher performance. IPC is determined by dividing the total number of instructions executed to run a program by the total number of cycles to run the program.
- IU:** Integer Unit. In the SPARC architecture, the integer controls program execution, initiates memory operations, and computes integer results.

## J

**JTAG:** Joint Test Access Group, IEEE 1149.1, serial scan test access facilities. The JTAG test access port gives a five wire test connection to many integrated circuits to facilitate board and system testing.

## K

**keeper:** See bus keeper.

## L

**latency:** The time from starting an operation until the result is available for use by another instruction. See also **throughput**.

**local register:** The registers in a register window that are not shared with either of the adjacent register windows and are, hence, local to the window. These registers can be used as private registers by a procedure.

**lock (in cache):** A term applied to cache blocks that can be locked in cache. Locked entries are never selected for replacement, so a locked block remains in the cache.

**locked operation:** On a bus, any sequence of bus accesses performed in such a way as to prevent any other bus master from accessing the bus during the sequence of accesses. The SPARC architecture has several atomic operations that may be implemented using locked bus operations. Each of the SPARC atomic operations requires one read and one write to be performed on the addressed location.



## M

**MEMOp:** Memory operation. Load and store instructions, including atomic load/store instructions, are MEMOps.

**memory reference:** An access to cache memory or main memory by a load or store instruction.

**message:** See **packet**.

**miss:** An attempt to access data in a cache that fails because there is no association between the address of the access and a valid entry in the cache. A load or fetch miss normally initiates an automatic read of the requested memory from main memory or the next level of cache. Once the read completes, the data may be accessed.

**MMU:** Memory Management Unit. A unit of a processor that performs functions related to memory address translation and protection.

**multiprocessing:** Using more than one processor in a computer system. See also **shared memory multiprocessing**.

## N

**normalized:** Cast into normal form. Floating-point numbers are usually normalized (see **denormalized**). The normalized format requires that the exponent be adjusted until the fraction can be represented with an implied 1 in the bit position beyond the highest order bit of the fraction.

**nPC:** Next Program Counter (PC). The address of the next instruction to execute. In an architecture with delayed control transfer instructions, a CTI changes the next value of nPC rather than the next value of PC. It is necessary for both PC and nPC to be saved on a trap or interrupt because, during the execution of the delay instruction after a DCTI, the nPC has the address of the destination of the control transfer.

## O

**out registers:** The registers of the register window that are the same as the in registers of a called routine after executing a SAVE. They may be used to pass arguments to a called routine and to receive returned values from the called routine.

**P**

**packet:** A message consisting of a header and sometimes data. A packet may be either a request packet or a reply packet. A request packet is sent to a bus slave. A reply packet is returned later.

**packet header:** The portion of a packet that contains addressing information, commands and other control information. The remainder of the packet contains its payload of data.

**packet-switched bus:** That which communicates messages containing requests and replies. The messages are called packets. The bus is free for other messages during the time between a request and its reply. A packet-switched bus requires more complicated interface controllers than the more widely used circuit-switched busses.

**page:** The smallest unit of memory address translation. A single entry in the TLB or page tables maps a range of addresses called a page. In the SPARC reference MMU, a page is 4KB.

**page table:** A collection of virtual to physical page address translations, each of which is called a page table entry (PTE). The SPARC Reference MMU uses page tables organized into a tree data structure in order to save space since most of the address space for a process is never valid.

**pipelining:** A technique for increasing the throughput of a processor by dividing the processing of every instruction into a few steps. The steps are then overlapped so that the first step of an instruction is processed at the same time as the second step of the previous instruction and the third step of the instruction before that and so on.

**pipeline stage:** Processor logic bounded by registers on each side and corresponding to one of the steps in the execution of an instruction.

**phase:** Clock phase, or one-half of a clock cycle. The phase corresponds either to the time when the clock is high or when it is low. Many actions in the SuperSPARC processor occur in particular clock phases.

**physical address:** An address used to access memory and I/O devices on the bus. The physical address is generated by the MMU from the virtual address calculated by the processor.

**physically addressed cache:** A cache memory where the address tag of words in the cache are physical rather than virtual addresses.

**prefetch:** A memory access performed because it is likely to be used in the near future. A prefetch may be either an instruction prefetch or a data prefetch. Prefetching is generally autonomous to program execution, allowing the processor to continue while the prefetch is performed.

**prefetch queue:** A FIFO buffer that holds instructions from the instruction cache before they are issued.

**privileged instruction:** That which generally accesses features that could compromise system security or reliability, and so are restricted to trusted supervisory software. The execution of a privileged instruction is restricted to supervisor mode.

**privileged mode:** See supervisor mode.

**program order:** The order in which the instructions in a program would be executed on a strictly sequential processor which completed each instruction before fetching the next. Such implementation techniques as pipelining, superscalar execution, speculative execution, and branch prediction all execute at least some portion of most instructions in orders other than program order. Within a processor, these variations from program order cannot be observed by the program, but I/O, memory and other processors may see some variations in order. SPARC's memory models allow the selection of the types of variations from program order that the program can tolerate.

**PSR:** Processor State Register. This register is defined as part of the SPARC architecture. It has a number of bits that show program execution status and others that control program execution.

**PSO:** Partial Store Ordering. One of the memory models selectable in a SPARC processor. In PSO, even though store operations are issued in program order, they may not complete in program order. If the program requires that some pair of store operations complete in program order, a special store barrier instruction (STBAR) should be placed between them. PSO is the highest performing of the SPARC memory models.

**PTE:** Page Table Entry. An entry in the page tables that contains a final address translation with permissions, as opposed to a PTP that contains only the address of another part of the page tables.

**PTP:** Page Table Pointer. An entry in the page tables that contains the address of the next portion of the tree-structured page tables to consult, rather than the final address translation and permissions as in a PTE.

**pure consistency operation:** A bus operation that transfers no data but affects the state of a cached block. A pure consistency operation is only needed to carry out the cache consistency algorithm, not to supply data.

**Q**

**queue:** Any buffer that stores entries on a first-in first-out (FIFO) basis. Examples are the SuperSPARC processor's store buffer and prefetch buffer.

**R**

**read ports:** The number of interfaces available for reading, and hence the number of concurrent read operations that can be supported on a memory or register file.

**region:** One of the large mapping sizes supported by the SPARC Reference MMU. A region is a 16 Mbytes of contiguous memory and is mapped by a single PTE in the level 1 page table.

**replacement:** A term applied to an entry in a TLB or cache memory when it is replaced when needed for another entry that must be brought into the TLB or cache. The choice of which entry to replace is called the replacement algorithm. If the entry being replaced is dirty, it must also be copied back.

**reset:** An operation that restores certain parts of the state of a device. On the SuperSPARC processor, a reset operation sets just enough state for the device to enter the reset handler reliably from any state. The reset operation can be initiated by an external signal, by the JTAG TAP controller, by error mode, or by completion of BIST.

**S**

**segment:** One of the large mapping sizes supported by the SPARC Reference MMU. A segment is 256 Kbytes of contiguous memory mapped by a single PTE in the level-2 page table.

**set associativity:**  $n$ -way set associativity. A term applied to a cache in which the data for some address might be found in one of a small number ( $n$ ) of blocks in the cache. Each address has a set of  $n$  cache locations with which it can be associated. The number of blocks in the cache is determined by multiplying the number of sets times the associativity.

**shared:** That state of a cache block in which one or more other caches in the system has a copy of it. **Shared** is the opposite of **exclusive**.

**shared memory multiprocessing:** A shared memory multiprocessing system has more than one processor sharing the same main memory. The processors can address the same memory and expect to see a consistent time series of values in the memory locations. See **cache consistency**.

**snooping:** On a bus, that quality of an interface that observes transactions for which it is neither the master nor the addressed slave. Snooping is an important part of many algorithms for maintaining cache consistency in bus-based systems.

**SO:** Strong Ordering. One of the SPARC memory models. All load and store operations under SO complete globally in program order. SO is the lowest performing of the SPARC memory models.

**software pipelining:** Spreading the execution of each loop iteration over several actual trips through the loop.

**SPARC:** Scalable Processor Architecture. SPARC is an open processor architecture controlled by SPARC International, an industry consortium. It is defined in a published specification, the *SPARC Architecture Manual*. The architecture and specifications are versioned. The SuperSPARC processor is conformant to SPARC version 8.

**SPARC Reference MMU:** The SPARC Reference MMU is a memory management unit architecture specified in *The SPARC Architecture Manual*.

**speculative instruction issue:** A term applied to an instruction that is issued speculatively before it is known whether program flow will reach the instruction. Any instruction issued speculatively may need to be aborted. In a sense, any instruction is issued speculatively if it is started before a previous instruction that might trap has completed without trapping. This term usually applies to instructions after a branch on either the taken or untaken path of program execution issued before the branch direction has been decided.

**SRAM:** Static Random Access Memory. SRAM's low access time makes it a desirable memory technology for many high speed applications. Another plus is that reads do not destroy the data as in many RAM technologies, so that reading a bit does not require re-writing it.

**SRMMU:** SPARC Reference MMU. See above.

**stall:** A cycle in which no stage of the pipeline advances. No instruction in the pipeline proceeds beyond its current step. No traps (even interrupts) can occur, and no results are stored. See also **bubble**.

**store-buffer:** A FIFO queue of memory store operations. Each entry has the data to be stored, the main memory address at which to store it, and such other information as the size of the store operation. The store buffer allows the store instruction initiating the store operation to complete while the store buffer manages the store operation as it gets sent to main memory.

**superscalar execution:** That which occurs when a processor issues more than one instruction at a time from a single program. These simultaneously issued instructions are executed concurrently.

**supervisor mode:** The mode in which a SPARC processor is said to be executing when  $PSR.S=1$ . In supervisor mode, privileged instructions may be executed. Supervisor mode and **user mode** are complementary; the processor always executes instructions in one mode or the other.

## T

**TBR:** Trap Base Register. This register is defined in *The SPARC Architecture Manual*. The trap base register locates the system trap table, which contains entries for the many trap types supported in the architecture.

**test access port (TAP) controller:** An internal sequencer that manages access to all JTAG test data registers.

**three-state buffer:** A three-state output buffer which may drive its output signal high or low, or may be in a high impedance state where it does not affect the output signal. Bi-directional pins usually have three-state buffers.

**throughput:** The rate at which data processing operations can be performed. Throughput can also be the reciprocal of the amount of time from starting an operation until another operation can be started. If the operation is pipelined, this time can be much shorter than the latency of the operation. For example, a floating-point multiply may have a three cycle latency but a one cycle throughput. This means that the multiplier can produce one product after three cycles and four products after six cycles. This multiplier can multiply to produce results at the peak rate of one every cycle. If it is operating at 50 MHz, its peak throughput is 50 million multiplies per second.

**TLB:** Translation Lookaside Buffer. The TLB is a cache of address translations that is part of the MMU. It is associative, based on the page number portion of the virtual address. When an association is found in the TLB for the page portion of the virtual address, the TLB produces the physical page number portion of the physical address and properties of the translation, including cacheability and access permissions.

**trap:** A vectored transfer of control to supervisor software through the trap table. Traps are caused by enabled exceptions, errors, resets, or interrupts.

**TSO:** Total Store Ordering. One of the SPARC memory models that allows store operations to be buffered and then completed in the system at a later time. In TSO, all store operations complete in the order in which issued, but load operations may complete before earlier stores, which might still be held in the store buffer. A load to an address that has a store pending in the store buffer will wait for that store to complete. The performance of TSO lies between that of SO and PSO.

**U**

**user mode:** The mode in which a SPARC processor is said to be executing when PSR.S=0. In user mode, any attempt to execute a privileged instruction generates a privileged instruction trap. User mode and **supervisor mode** are complementary; the processor always executes instructions in one mode or the other.

**V**

**virtual address:** An address as generated by the processor for access to data and instructions. The virtual address is translated by the MMU into a physical address that is used to access memory and I/O devices. See also **page**.

**W**

**WIM:** Window Invalid Mask. This register is defined as part of the SPARC architecture. It marks register windows that are invalid. SAVE or RESTORE instructions that change the current window to an invalid window cause a **window\_overflow** trap or a **window\_underflow** trap, respectively.

**write-back cache:** A cache in which store operations are performed on data without also updating memory. When a block in a write-back cache is replaced, it must be copied back to memory. A block that has been modified in cache is called **dirty**.

**write ports:** The number of interfaces available for writing, hence the number of concurrent write operations that can be supported on a memory or register file.

**write-through cache:** A cache in which store operations on data update both the cache and main memory. When a block in a write-through cache is replaced, it can be invalidated without the need to copy back to memory, since memory already has the new data value. Cache blocks in a write-through cache never become **dirty**.

**X****Y****Z**





# Index

---

## A

- aborted instruction, 5-2
- AC, 10-22
  - bit, 10-37, 14-25
- ACC, field, 9-33
- access
  - bus error code, 9-33
  - exception, 5-5
  - permission code, 9-4, 9-38
  - to debug features, 15-6
- access exception, 5-5
- access type, 9-31, 9-33
- accrued exception field, 4-15
- ACTION ASI registers, 15-8
- action on breakpoint event register, 12-12
- ACTION register, 12-12, 15-3, 15-7, 15-9, 15-11
- ACTION.BCIPL, 12-12, 15-5
- ACTION.IEN\_DBK, 15-5
- ACTION.MIX bit, 13-4, 13-5
- action-on-event register, 15-3
- ADD instruction, 5-23, 5-24
- additional integer, 5-6
- address
  - breakpoint, 15-3
    - facilities, 15-2
  - commands, 20-6
  - computation, 5-10
  - cycle, 17-7
  - decoder, 20-6
  - dependencies, 5-6
  - format, 10-15
  - high byte command, 20-6
  - low byte command, 20-6
  - middle byte command, 20-6
  - operands, 5-5, 5-6
  - space, 2-2
    - implicit alternate*, 22-22

- space identifier, 2-3, 2-6
- tag, 16-9
- translation, 9-3
- translation modes, 9-19
- addressing conventions, 2-5
- AE error, 16-43
- aexc, 4-15
- allocate policy, 17-13
- altered non-emulation PC pair, 22-18
- alternate
  - address space, 22-22
  - cacheable bit, 8-6, 9-23, 10-22, 10-37
  - space atomics, 8-5
- ALU, 4-4, 5-5, 5-6, 6-19
- ancillary state register, 4-10
- application code, 21-2
- approximate latencies for each emulator primitive, 22-30
- arbiter, MBus, 17-2
- arbitration, VBus, 18-8
- arbitration priorities, 19-44
- architecture, 3-7
- arithmetic, 2-7
- arithmetic/logical/shift instruction, 2-5
- ASI, 2-3, B-1
  - 0x39, 13-8
  - control spaces, 10-38
  - memory references, 21-11
  - operations, 8-6
  - values, B-1
- ASR, 4-10
- assembly language, 21-2
- asynchronous scan interface, 3-5
- AT, 9-31
- atomic operations, 8-4, 10-35

## **B**

- BA branching, 15-11
- BA instruction, 15-6
- backplane-level JTAG busmaster, 21-22
- bandwidth, 16-4
- basic instruction operations, 2-2
- BCIPL, 15-13
- BCLK, 17-42, 23-7
- BCLK and PCLK relationship, 23-7
- big-endian architecture, 2-5

- binary floating-point arithmetic, 2-4
- BIST, 13-2, 13-7, 21-11
  - ASI operation, 13-8
  - coverage, 13-7
  - JTAG-initiated, 21-11
  - long version, 13-7
  - mechanism, 21-11
  - operation warnings, 13-9
  - operations, 21-11
  - register domain, 21-9
  - register domains, 21-7
  - Reset, 14-5
  - sequencer, 21-9, 21-11
  - short version, 13-7
  - .SIGNATURE register, 13-7
  - .STATUS, 13-8
- BKS.DBKIS bit, 15-5
- BKV, 15-8
- block
  - number field, 16-17
  - read, 8-10
  - read facility, 16-30
  - write, 16-30
- board-level JTAG busmaster, 21-22
- BootBus, 20-1
  - address, 20-6
  - address decoding, 20-5
  - commands, 20-6
  - controller, 20-6, 20-7
  - example transactions, 20-9
  - interface, 16-11
  - introduction, 20-2
  - signals, 20-3
  - transactions, 20-6
- boot mode, 9-23, 13-4
- boot mode bit, 12-9, 13-6
- boot mode/local bus indicator, 17-10
- bootstrap loading, 20-1
- boundary scan, software-controlled, 21-2
- boundary scan map (Viking), 21-10
- branch, 5-19
  - couple, 5-21
  - direction, 6-2
  - frequency, 6-2
  - instruction, 22-16
  - on floating-point condition codes instruction, 5-14
  - performance, 6-2
- breakpoint
  - ACTION register, 15-12

- address compare mask, 22-25
- and counter-interrupt level, 12-12, 15-7
- code address, 22-25
- code and data address, 22-25
- control register, 15-8
- control register (BKC), 12-12
- control registers, 15-7
- data address 22-25
- hardware, 3-5
- mask register, 15-8
- status register, 15-9
- value register, 15-7
- BSCAN, 21-7, 21-10, 21-15
- BT, 9-24
- bubble, 6-10
- buffering, 3-4
- buffers and synchronizers, 16-11
- built-in self-test, 13-2, 13-7, 21-2
- burst mode access bit, 10-41
- burst read
  - hit, 18-22
  - miss, 18-24
- burst transaction, 8-10
- bus
  - commands, 19-23
    - data commands, 19-23*
    - tag commands, 19-32*
  - command logic, 16-9, 16-10, 16-11
  - connection, 16-4
  - cycle waveforms, 19-49
  - error, 9-30, 12-15
  - error response, 17-39, 17-47
  - grant, 17-34
  - keeper, 18-10, 18-18
  - master, 17-1
  - ownership, 19-46
    - default grantee, 19-47*
    - no default grantee, 19-47*
  - protocol, 19-15
    - cycles, 19-15*
    - packet detection, 19-16*
    - packets, 19-15*
    - transactions, 19-16*
  - slave, 17-1
  - snooper, 17-1
  - snooping, 3-5
  - transaction, VBus, 18-6
  - watcher, 16-3, 19-6, 19-46
- BYPASS, 21-7, 21-10

- instruction, 21-14
- scan chain, 21-10
- byte
  - access, 2-5, 13-8
  - reference, 5-8

## **C**

- C bit, 10-37, 14-25
- cache
  - arrays, 13-7
  - block, 16-14
  - block shared, 17-8
  - coherence protocols, 17-2, 17-3
  - coherent support, 3-4
  - column redundancy repair circuits, 13-5
  - consistency, VBus, 18-6
  - consistency protocol, 16-9, 17-16, 19-21
  - consistency state, 17-18
  - control registers, 2-6
  - controller, 13-9, 16-9
  - disable read, 18-27
  - disable write or non-cacheable write, 18-35
  - enable bit, 13-10, 17-41
  - lookup, 5-5, 5-10
  - miss, 5-5
  - miss penalties, 6-2
  - RAM, 3-5
  - redundancy logic, 13-6
  - size bit, 16-15
  - split I & D, 2-7
  - sub-block states, 19-21
  - XBus configuration, 19-21
- cacheability, 10-7, 10-21
- cacheable
  - bit, 10-41
  - page, 9-4, 9-37
  - single read hit, 18-19
  - single read miss, 18-20
  - write hit, 18-28
- cached entries, 3-7
- caches, 14-22
- caches/store buffer, 10-1
- CALL instruction, 5-23
- calls, 5-19
- CAPTURE operation, 21-5, , 21-6, 21-8, 21-11
- captured non-emulation PC pair, 22-18
- cascade
  - conditions, 5-4

- instruction group, 6-14
- CBFEN, 15-9
- CBKEN, 15-9
- CBKFS, 15-10
- CBKIS, 15-10
- CBKM, 22-8, 22-9
- CC error, 16-43
- CCNT, 15-10
- CCNTEN, 15-11
- CCOP field, 16-43, 16-44
- CE bit, 17-41
- cexc, 4-12, 4-15
- chip pins, 21-10
- CI
  - operation, 17-44
  - transaction, 17-14, 17-18, 17-25, 17-26, 17-28, 17-36, 17-37, 17-41, 17-44, 17-45
- CID, 21-7
- CID primary register's scan chain, 21-10
- circuit-switched bus, 19-4
- clean stable clock sources, 23-6
- CLK, 17-13
- clock
  - clocking, 23-1
  - delayed, 2-2
  - domain, 16-11
  - duty cycle, 23-6
  - external, 23-2
  - feedback loop, PLL, 23-2
  - input, 23-1, 23-2, 23-6
  - PLL, 23-3, 23-4
  - jitter, 21-13
  - pin, external, 23-2
  - requirements, iv, 23-1
  - routing, internal, 23-2
  - signal, 23-2
  - skew, system, 23-1
  - sources, 23-6
- code
  - address, 15-2
  - and data address breakpoints, 22-25
  - breakpoints, 22-25
  - generated breakpoint, 12-12
  - generation, 6-6
  - generation issues, 6-7
  - performance, 6-3
  - profiling, 15-2
- coherence algorithms, 3-5
- coherent

- cache multiprocessing, 17-2
- invalidate transaction, 17-14
- read and invalidate transaction, 17-14
- read transaction, 17-14
- write transaction, 17-14
- command
  - logic, 16-9
  - word signals, MBus, 17-9
- common
  - emulator
    - functions*, 22-19
    - primitives*, 22-19
  - level-two
    - TDI*, 21-22
    - TDO*, 21-22
- component ID, 21-10, 21-15
- compound emulation protocol commands, 22-17
- compound MCMD mode, 22-18
- condition codes, 4-4, 5-6
- conditional branches, 5-6, 5-18
- configuration space, 16-29
- context numbers, 9-3
- context register, 9-3, 9-24
- context table pointer register, 9-24
- control register, 9-22, 14-21
- control space, 16-17, 16-21
  - access, 16-12
  - access error, 9-27, 9-30
  - annulled, 2-7
- control transfer, 2-5, 5-4
  - instruction couple, 2-7, 5-21
  - instructions, 22-16
  - mechanism, 5-19
  - operation, 12-15
  - delayed, 4-9
- coprocessor, 2-2
  - disabled trap, 12-14
  - instructions, 4-5
  - interface, 12-14
  - operate, 2-5
  - support, 2-2
  - SPARC architecture, 2-4-2-5
- copy-back
  - cache, 5-6
  - data, 10-35
  - mode, 10-21
  - policy, 5-6
  - write-allocate caching policy, 10-35
- copy-out, 12-12



- counter, 21-2
  - cycle, 3-5
  - instruction, 3-5
  - breakpoint, 15-3, 15-7
    - control, 15-11
    - registers, 12-12
    - status, 15-11
    - value, 15-10
  - generated breakpoint, 12-12
- CP, 2-2, 2-3, 2-4
  - disabled trap, 4-5, 12-14
  - error, 16-43, 16-44
- CP\_instruction trap, 12-14
- CPI, 3-7
- CR
  - operation, 17-23, 17-25, 17-38
  - transaction, 17-8, 17-14, 17-15, 17-16, 17-23, 17-26, 17-27, 17-35, 17-36, 17-37, 17-43, 17-44, 17-45
- CRI, 17-28
  - transaction, 17-8, 17-14, 17-15, 17-25, 17-26, 17-35, 17-36, 17-37, 17-41, 17-43, 17-45
- CS bit, 16-15
- CSPACE, 15-9
- CTI, 5-21, 6-22, 10-11
- CTRC, 15-11
- CTRS, 15-11
- CTRV, 15-10
- CTRV.CCNT, 15-11
- CTRV.ICNT, 15-11
- current
  - exception field, 4-12
  - window pointer, 2-3, 2-11
- CWP, 2-3, 4-3, 4-5, 4-6, 5-23, 5-24, 12-14
  - change, 5-23
  - pipeline, 5-23
  - value, 5-23
- cycle
  - count underflow, 15-2
  - counter, 3-5
    - breakpoint, 22-28
    - interrupt, 15-12
- cycles, 19-15

## D

- d register addresses, FPU operation, 11-3
- daisy chains, parallel, 21-22
- data
  - register dependencies, 6-2
  - access

- errors, 9-27
  - exception trap, 12-15
- address, 15-2, 22-9, 22-25
- cache, 5-5, 5-6, 5-11, 10-26, 13-5, 14-12
  - consistency, 10-26
  - control, 10-29
  - data, 10-33
  - enable, 9-24
  - flash clear, 10-29
  - diagnostics, 10-29
  - high integration, Viking introduction, 3-4
  - support routines, 14-17
  - tags, 10-30
- commands, 19-23
- formats, 2-4
- forwarding, 5-6
- forwarding, 6-13
- generated breakpoint, 12-12
- operands, 5-5
- store error handler code, 10-37
- store error trap, 12-13
- data\_store\_error, 12-13
- datascan instruction, JTAG instruction register, 21-16
- DBFEN, 15-9
- DBKFS, 15-10
- DBKIS, 15-10
- DBKM, 22-8, 22-9
- DBREN, 15-9
- DBWEN, 15-9
- debug, 15-2
  - analysis, 3-5
  - interrupt, 15-2
- default grantee, 19-46, 19-47
- deferred floating-point traps, 5-13, 12-15
- delay
  - group of a branch, 6-22
  - instruction, 4-9
  - instructions allocation, 6-7
- delayed control transfer, 2-2, 4-9
- demand fetch, 10-12
- DeMap operation, 8-8
- dependent FPop, 5-14
- destination
  - address, 16-30
  - data formats, 11-9
  - field, 2-5
  - register, 2-5, 6-13
- device manufacturing correctness, 21-2

- diagnostic
  - emulation, 15-1, 22-1
  - operation (emulation), 15-1, 22-1
  - registers, 13-8, 15-7
- dirty bit, 10-32
- divide, 5-14
- divide by zero trap mask, 4-13
- double floating-point registers, 6-4
- double-precision
  - dividend, 4-8
  - operand, 4-11
  - value, 2-3
  - values, 2-3
- double-word, 2-5, 13-8, 16-5
  - load, 4-2
  - reads, 22-22
  - reference, 5-8
  - references, 12-15
  - size, 22-25
- drain pointer, 10-43
- DRCAPTURE, 21-6, 21-10
- DRSHIFT, 21-10, 21-13
- DRUPDATE, 21-10
- dual-port register files, 16-11
- duty cycle
  - clock, 23-6
  - input, 23-6

## E

- E-cache, 16-4, 16-14, 17-18, 17-41
  - data access, 16-15
  - tag entry, 16-18
- E1, 5-6
- EC, 4-5
- EC bit, 4-5
- ECC errors, 9-30
- ECC problem, 12-13
- ECHOTMR, 22-8
- EF bit, 2-4, 4-5
- eight-byte boundaries, 2-5
- eight-doubleword entry store buffer, 3-4
- emulation, 21-12
  - command and instruction, 22-5
  - counter status register, 15-12
  - data memory reference instruction, 22-16
  - data out, 22-7

- entering, 22-14
- entry, 22-14
- execution, faults during, 22-15, 22-16
- exit PC/NPC registers, 22-12
- instruction, 21-2, 22-16, 22-17, 22-18, 22-30
  - last*, 22-18
  - legal and illegal*, 22-16
  - previous*, 22-18
  - scan in*, 22-30
  - scan out*, 22-30
  - sequence*, 22-12, 22-19, 22-20
- mode, 13-2, 22-11, 22-15, 22-16, 22-17, 22-18, 22-19, 22-25
- operation, 22-15, 22-20
- primitive, 22-30
- register domain, 21-7, 21-9
- registers, 22-4, 22-12
- remote, 22-18
- sequence, 22-17, 22-19
- session, 22-18
  - first*, 22-18
  - second*, 22-18
- software, 22-23
- status, 22-8
- strategy, 22-2
- EN, 9-24
- enable
  - coprocessor bit, 12-14
  - floating-point bit, 2-4
  - trap field, 5-25
  - traps bit, 12-11
- end reset, 14-27
- entering emulation, details, 22-14
- entry number, 10-40, 10-42
- entry type, 9-5, 9-38
- EPROM, 20-1
- ERR field, 16-44
- ERRMODE, 22-10
- ERR, 16-43, 16-44
- ERR bit, 16-43
- ERRMODE bit, 22-10
- error
  - code, 16-44
  - handling, MXCC, 16-36
  - mode, 9-27, 12-10, 22-16, 22-17, 22-19
    - bit*, 13-5, 13-6
    - condition*, 12-14
    - reset*, 14-4
    - reset taken*, 9-30
  - register, MXCC, 16-42

- reply, 16-44
- reporting, VBus, 18-6
- traps, 12-9
- ERROR1
  - acknowledgement, 17-31
  - response, 17-39, 17-47
- ERROR2
  - acknowledgement, 17-28
  - response, 17-39, 17-47
- ERROR3
  - reply, 17-40, 17-47
  - response, 17-39, 17-47
- ESB, 15-5
- ESB pin, 15-3, 15-13, 15-15, 22-26
- ET, 4-5
- ET bit, 12-9, 16-43, 16-44
- even-odd pair of registers, 2-3
- event-dependent bit, 15-5
- exception, 9-14, 12-7, 15-2
  - handling, 5-24, 12-7
  - mode, 4-16
  - next program counter, 12-7
  - pipeline, 5-24
  - program counter, 12-7
  - sources, 5-5
  - and the pipeline, 5-24
- execution pipeline, blocking, 10-35
- execution stages, 5-5
- existing code performance, 6-3
- exit, 22-17
- extended cache, 16-14
- extended opcode space, implementation-dependent, 4-10
- external
  - analysis equipment, 3-5
  - bus
    - error, 9-33
    - request, 16-9
  - cache, 3-4, 5-6, 16-4
    - controller, *iv*, 16-1
    - RAM, 16-4
    - support, *high integration, Viking introduction*, 3-4
    - tags, 16-17
  - clock, 23-2
  - decoder, 20-7
  - device, 15-15
  - interrupt requests, 12-11
  - monitors, 15-14
  - strobe pin, 15-3

# F

- FAR, 12-13
- fast load and store instructions, 3-7
- fault
  - address register, 9-34, 12-13
  - address valid bit, 9-34
  - status register, 9-34
  - type, 9-32
  - during emulation execution, 22-16
- faulty emulation status indication, 22-16
- FBfcc, 4-14, 5-14, 5-16, 11-2
- FCmp instruction, 4-14, 5-14, 6-11
- FCmp-FBfcc pair, 6-11
- FCMPE instruction, 4-14
- FE stage, 5-15
- feedback loop, clock, PLL, 23-2
- fetch, 5-4
- FIFO, 10-11
- FIFO queue, 3-4, 5-16, 16-11
- file write, integer register, 22-21
- fill pointer, 10-43
- filter circuit, recommended, 23-3, 23-4
- first emulation session, 22-18
- flash clear, 10-17
- floating-point, 5-16
  - arithmetic
    - binary, 2-4*
    - IEEE Standard 754, 2-2*
    - IEEE Standard 754-1985, 2-4*
    - floating-point arithmetic conditions, 12-15*
    - floating-point arithmetic functions, single- and double-precision, high-performance, 3-4*
    - floating-point arithmetic instructions, 5-16*
  - branches, 4-14
  - code issues, 6-9
  - compares, 5-16
  - converts, 5-16
  - control register, reading, 22-23
  - data formats, 2-4
  - disabled trap, 12-14
  - events, 5-13, 5-16
  - exception, iv, 11-1
    - details, 11-4*
    - exception trap, 12-15*
    - exception trap type, 4-14*
  - f registers, register summary, 4-11
  - implementation, 3-8
  - instruction, 2-4, 4-5, 11-2

- most recently executed*, 4-15
- instruction sets, 2-4
- LD/ST, 6-19
- load/store instructions, 2-4
- moves, 5-16
- operate, 2-4, 2-5
  - instructions, SPARC architecture*, 2-8
- operation scheduling, 6-2
- operations instruction, 5-6, 5-13, 5-16
- pipeline, 5-13
- queue, 4-16, 5-16, 12-15
  - FPU operation*, 11-8
  - register summary*, 4-16
  - interface, iv*, 11-1
- register, 2-2, 2-3, 5-13
  - dependency*, 6-8, 6-13
  - read*, 22-23, 22-25
  - register file*, 2-2
  - state*, 4-12, 22-25
  - write*, 22-24
- transfers, 3-7
- traps, deferred, 5-13
- unit (FPU), 2-2, 2-2, 2-3, 2-4, 3-4, 4-5, 5-6, 5-16, 14-27, 22-11
  - accrued exception field*, 4-15
  - high integration, Viking introduction*, 3-4
  - operation*, 11-1
  - register*, 2-3, 4-13, 5-16
  - SPARC architecture*, 2-3-2-4
- flow control, 19-45
  - bus watchers, 19-46
- flow of execution, 22-16
- flush (IFLUSH), 7-5
- FLUSH instruction, 2-7, 10-13
- FMULS, 6-9
- format 1, A-4
- format 2, A-5
- format 3, A-7
- four-byte boundaries, 2-5
- FP
  - data dependencies, 6-8
  - exception, 22-11, 22-25
  - pipeline, 5-6
  - queue, 22-25
  - register dependencies, 6-8
- fp-disabled trap, 4-5
- fp\_exception trap, 4-12, 4-15
- FPev instruction, 5-13, 5-16
- FPop, 2-4, 5-13, 22-25
  - dependent, 5-14

- instruction, 4-12, 4-13, 5-16
- latency, 6-10
- FQ, 22-11, 22-25
- FQE, 22-11
- FRD stage, 5-15
- freq dependency, 6-12, 6-13
- FSR, 4-12, 4-16, 11-3, 22-25
- FSR.IMPL field, 11-3
- FSR.QNE, 5-16
- FSR.VER field, 11-3
- ftt, 4-14
- full testability, high integration, Viking introduction, 3-6
- futurebus, 17-2
- FWB stage, 6-13

## G

- general-purpose
  - integer registers, 2-2
  - general-purpose programs, 2-3
- gradual underflow, 2-4

## H

- half cache bit, 16-15
- half-word, 2-5
  - access, 2-5, 13-8
  - reads, 22-22
  - reference, 5-8, 12-15
  - (16-bit) access, 2-5
- hardware
  - assisted software memory-scrubbing scheme, 16-30
  - breakpoints, 3-5
  - interrupt requests, 12-11, 12-16
  - replacement algorithm, 10-17, 10-33
  - reset, 12-13, 13-2, 13-5, 13-6, 14-6, 22-9
  - reset requirements, 13-5
  - reset response, 13-5
  - /software page table consistency, 8-8
  - use of page tables, 8-8
- Harvard architecture, 2-7
- HC bit, 16-15
- heinous modes, 13-6
- high byte, 20-6
- high integration, Viking introduction, 3-2
- high-order bits, 5-5
- high-performance floating-point arithmetic functions, 3-4
- highly dependent operation, 5-14



hit

criteria, 9-11

ratio, 16-23

## I

I&D memories, 2-7

icc, 4-4

icc.c, 4-5

icc.n, 4-4

icc.v, 4-4

icc.z, 4-4

icc-modifying instruction, 4-4

ICNT, 15-10

ICNTEN, 15-11

IDIV, 5-17, 7-3

WRPSR, 7-4

idle command, 20-6

IEN\_CBK, 15-13

IEN\_DBK, 15-13

IEN\_ZCC, 15-13

IEN\_ZIC, 15-13

IFLUSH instruction, 7-5, 14-26, 15-6

illegal

instruction trap, 2-4, 4-2, 4-5, 12-14, 12-16

instructions, 22-16

opcode, 12-14

operations, 22-17

implementation number, 9-22, 22-19

implementation-dependent, 2-3

improperly aligned address, 2-5

IMUL, 5-17, 7-2

IMUL/DIV operations instructions, 5-16

incoming processor commands, 16-9

independent scan rings, 21-12

inexact trap mask, 4-13

infinitely exact result, 4-13

inhibit transaction, 17-14

initiating BIST, 13-7

input

clock, 23-2, 23-6

duty cycle, 23-6

queue, 16-10, 16-11

input/output

RDASR, SPARC architecture, 2-10

SPARC architecture, 2-10

instruction

FLUSH, 2-7

Ticc, 2-11

access errors, 9-26

access exception trap, 12-13

access faults, 9-34

cache, 6-19, 10-5, 10-12, 10-13, 10-18, 13-5, 14-11

*high integration, Viking introduction, 3-4*

*consistency, 10-13*

*controls, 10-14*

*data, 10-17*

*diagnostics, 10-14*

*enable, 9-24*

*replacement policy, 10-9, 10-22*

*support routines, 14-14*

*tags, 10-15*

count underflow, 15-2

counter, 3-5

counter breakpoint, 22-28

execution, 2-3

fields, instruction summary, A-2

formats, 2-2

grouping, 6-16

*classes of rules*

*exceptions, 6-16*

*split after, 6-16*

*split before, 6-16*

operands, 2-3

ordering, 6-2

prefetching, 10-11, 10-12

queue, 6-19, 10-11

queue prefetch buffer, 5-4

register, 21-8, 22-2

sets, floating-point, 2-4

space, RDASR-reserved, 4-10

summary, instruction fields, A-2

instruction\_access\_exception, 12-13

instructions

arithmetic, SPARC architecture, 2-7

control transfer

*annulled, SPARC architecture, 2-7-2-8*

*delayed, SPARC architecture, 2-7-2-8*

*SPARC architecture, 2-7-2-8*

coprocessor operate, SPARC architecture, 2-8

floating-point operate, SPARC architecture, 2-8

flush (IFLUSH), 7-5

integer divide (IDIV), 7-3

*write PSR (WRPSR), 7-4*

integer multiply (IMUL), 7-2

load/store

- alternate, 2-6-2-14*
  - SPARC architecture, 2-5-2-7*
- logical, SPARC architecture, 2-7
- memory access, SPARC architecture, 2-5-2-7
- per cycle, 3-7, 6-3
- SETHI, SPARC architecture, 2-7
- shift, SPARC architecture, 2-7
- signal user emulation request (SIGM), 7-7
- SPARC architecture, 2-5
- state register access
  - ancillary state, SPARC architecture, 2-8*
  - SPARC architecture, 2-8*
- store barrier (STBAR), 7-6
- tagged arithmetic, 2-7
- that modify the condition codes, 4-4

integer

- arithmetic units, 6-5

integer

- divide, 7-3, 7-4, 11-7, 12-14
  - by zero trap, 12-16*
  - instruction, 3-4, 12-14, 12-16*

integer

- LD/ST, 6-19
- load and store instructions, 2-5
- multiply, 4-8, 7-2, 12-14
  - FPU operation, 11-7*
  - instructions, 3-4*
  - integer divide operations instructions, 5-16*
- register, 22-20, 22-21
  - file read, 22-20*
  - file write, 22-21*
  - state, 22-21*
- state
  - register, 22-21*
  - read, 22-21*
  - write, 22-21*
  - PSR, 22-21*
  - TBR, 22-21*
  - WIM, 22-21*
  - Y, 22-21*
- unit, 2-2
  - SPARC architecture, 2-3*
  - SPARC-compatible, 3-3*
  - r registers, 4-2*

interconnect testing, 21-2

interface logic, 16-10

internal

- BIST operation, 13-7
- cache, 5-6
- clock routing, 23-2

- error code, 9-33
- interrupt sources, traps, 12-12
- logic, 23-2
- routing delay, 23-2
- sequencer, 21-4
- INTERNAL\_SCAN, 21-13
- interrupt, 5-25
  - request, 2-11
  - command, 20-7
  - generation register, 16-34
  - latency, 12-11
    - traps*, 12-11
  - levels, traps, 12-16
  - mask register, 16-33
  - pending
    - clear register*, 16-34
    - register*, 16-33
  - request level, 5-25, 17-9
- interrupted program counter, 5-25
- introduction
  - JTAG serial scan interface, 21-2
  - MultiCache Controller, 16-2
  - to SuperSPARC, 1-1, 1-2
  - traps, 12-2
  - VBus, 18-2
- intscan instruction, JTAG instruction register, 21-16
- invalid
  - address error code, 9-32, 9-33
  - operation
    - exception*, 4-13
    - trap mask*, 4-13
  - register window, 12-14
- invalidate transaction, 17-14
- inverted parity, 13-9
- IPC, 3-7, 6-3
- IPND, 22-9
- IQ, 10-11
- IR, 21-8, 21-10, 22-4
  - register, 22-4
  - scan chain, 21-3
  - scan ring, 21-8
  - update register, 21-8
- IRL, 5-25
- IRUPDATE state, 21-8
- IU, 2-2, 2-3, 2-4, 5-5, 5-16
  - pipeline, 22-6
  - register, 2-3

## **J**

- JMPL, 5-22, 5-26, 6-20, 6-22
  - branching, 15-11
  - couples, 6-22
- JMPL/RETT pairs, 6-22
- JTAG, 22-17
  - busmaster, 21-4
    - backplane-level*, 21-22
    - board-level*, 21-22
  - CAPTURE operation, 21-6
  - controller
    - second-level*, 21-22
    - third-level*, 21-22
  - emulation, 3-5
  - instructions, multichannel controller, 21-14
  - interface, 13-2, 13-7, 21-3, 23-2
  - IR, 21-4, 22-2
  - MCMD scan register, 15-7
  - MCMD.INITM bits, 15-8
  - MDIN, 22-12
  - MDOUT, 22-12
  - mechanism, 21-2
  - memory references, 21-11
  - operations, 21-5
  - register domains, 21-7
  - reset requirements, 21-4
  - scan chain element, 21-6
  - scan chain register elements, 21-10
  - serial scan interface, 21-1
  - serial scan interface mechanism, 21-3
  - SHIFT operation, 21-6
  - status, 21-2
  - TAP, 22-6, 22-8
  - TAP controller, 21-4, 21-8, 21-11, 23-2
  - tap controller, 22-14
  - TAP controller reset, 22-7, 22-9, 22-10, 22-11
  - TDR Emulation Registers, 22-4
  - TDR scan chain, 22-4
  - TDR scan operation, 22-4
  - UPDATE operation, 21-6
  - accessible serial scan chains, 21-7
  - initiated BIST, 13-7
- jumps, 5-19

## **L**

- large-windowed register file, 2-2
- last emulation instruction, 22-18
- last pipe stage, 12-8

- latching, 5-4
- latency, 6-10
- LCMDS, 20-6
- LDA instructions, 22-12
- LDATA, 20-6
- LDD instruction, 5-24
- LDF/STF registers instruction, 5-16
- LDFSR instruction, 4-12, 4-14
- LDFSR/STFSR register instructions, 5-16
- LDST (load and store), VBus transactions and waveforms, 18-43
- LDSTUB instruction, 8-4, 15-3, 17-34, 17-42
- LDSTUBA instruction, B-1
- least significant bit, 4-2
- legal and illegal emulation instructions, 22-16
- level, 9-31
- level 2
  - signals, 17-2
  - TDI, common, 21-22
  - PTP2 cache, 9-10
  - TDO, common, 21-22
- linear mapping, 9-6
- linked list traversal, 6-15
- load, 5-14
- load alternate, 8-6
- load and store alternates, 8-6
- load and store instructions, 3-7, 2-5, 5-16
- load operation, 5-8, 12-15
- local flush
  - operation, 8-9
  - transactions, 8-9
- lock bit field, 8-4, 17-42, 10-10, 10-24
- lock indicator, 17-11
- logic, internal, 23-2
- logic analyzer, 15-15
- LONG\_BIST, 13-7, 21-7
- low byte, 20-6
- low order address bit, 5-5, 12-15
- low-speed peripherals, 20-1
- LDA/STA, 8-6

## M

- MACK, 22-8
- manufacturing fault coverage, 21-2
- mapping, linear, 9-6

- master, 17-2
- maximum interrupt latency, 12-12
- MBSEL pin, 13-12, 17-41, 20-1
- MBus, 17-1
  - arbiter, 17-2
  - busy, 17-8
  - clock frequency, 17-4
  - command word signals, 17-9
  - configuration, 16-9
  - configurations, Viking, 17-4
  - error, 17-7
  - grant, 17-8
  - level 2, 16-4
  - master clock, 17-6
  - mode, 9-24
  - mode, 5-6
  - module, 24-1
    - mechanical information, 24-9*
  - module identifier, 17-10
  - module schematics, 24-3
  - operation, 17-13
  - overview, 17-2
  - ready, 17-7
  - reference clock, 17-13
  - request, 17-8
  - retry, 17-7
  - signals, 17-6
  - standard, level 2, 3-4
  - system, full module, 24-2
  - timing summary, 17-48
  - transactions, 10-28
- MBus/XBus interface, 16-10
- MXCC, 13-9, 16-1, 23-4, 23-6
  - basic functionality, 16-8
  - BIST register, 16-23
  - block copy, 16-30
  - block diagram, 16-8
  - built-in self-test register, 16-23
  - consistency, 17-18
  - control register, 16-24
  - Error Handling, 16-36
  - error register, 16-42
  - internal registers, 16-21
  - interrupts, 16-33
  - MBus port register, 16-29
  - pins, D-2
  - reset register, 13-10, 13-12, 16-28
  - SRAM access, 18-49
  - status register, 16-26
  - synchronous and asynchronous operation, 23-7

- system reset, 13-10
- MCCCR.PE, 13-9, 14-25
- MCI, 21-7, 21-12, 22-17
  - register, 22-9
  - scan chain, 21-9, 21-13
  - TDR, 22-19
- MCI.MINST register, 22-17
- MCMD
  - mode, compound, 22-18
  - register, 22-19
- MCMD.INTM bit, 15-5
- MCMD.MENTER, 22-8, 22-9, 22-14, 22-19
- MCNTL, 9-22, 13-9
- MCNTL.AC, 14-25
- MCNTL.AC bit, 8-6, 10-37, 13-4
- MCNTL.BT bit, 12-9, 13-4, 13-6
- MCNTL.MB bit, 10-21
- MCNTL.NF bit, 9-20, 10-37
- MCNTL.PE, 13-9, 14-25
- MCNTL.PE bit, 13-9
- MCNTL.PSO bit, 8-3
- MCNTL.SB, 8-2
- MCNTL.SB bit, 8-3, 10-42
- MCNTL.SE, 10-27
- MCNTL.SE bit, 10-13
- MCNTL.TC, 14-25
- MCTP, 9-24
- MDIAG.BKC register, 12-12
- MDIN, 21-7, 21-12, 22-12, 22-19, 22-23, 22-25
  - register, 22-12, 22-23
  - scan chain, 21-9, 21-13
- MDOUT, 21-7, 21-12, 21-13, 22-7, 22-12, 22-25
  - port, 22-13
  - register, 22-13, 22-20, 22-21, 22-22, 22-23
  - scan chain, 21-9, 21-13
- mechanical information, MBus module, 24-9
- memories, separate I&D, 2-7
- memory, 2-4, 5-2
  - addressing, conventions, 2-5
  - alignment restrictions, 2-5
  - access, instructions, SPARC architecture, 2-5
  - address, 2-4
  - address and data, 17-6
  - address not aligned trap, 12-15
  - address register ports, 6-19
  - address strobe, 17-7
  - and I/O transactions, VBus, 18-7



- controllers, 17-2
- inhibit, 17-8
- model, 2-2, 8-1
  - SPARC architecture, 2-9*
  - support, 8-11*
- read, 22-22
- reference, 5-4, 4-16, 5-8, 6-14, 8-7
- state, 21-2
- write, 22-23
- memory-mapped emulation counter status register, 15-12
- memory-reference patterns, 6-2
- MENTER register, 22-17
- message priority detection, 19-48
- MEXEC, 22-6, 22-7, 22-17, 22-18
- MEXIT bit, 22-7, 22-10, 22-17, 22-18
- MFAR, 9-29, 9-34, 9-35, 22-10
- MFSR, 9-26, 9-28, 9-29, 9-34, 14-22, 22-10
  - error bits, 10-37
  - fault status register, 9-20
  - register description, 9-30
  - timing and operation, 9-28
- MFSR.CS bit, 9-27
- MFSR.CS status bit, 9-27
- MFSR.EM bit, 13-5, 13-6
- MFSR.FT bit, 9-29
- MFSR.FT field, 9-26, 9-28
- MFSR.OW, 9-29
- MFSR.SB bit, 9-27
- MFSR.SB error bit, 9-28
- MFSR.UC bit, 9-30
- MID, 17-10
- middle byte, 20-6
- MIDONE bit, 22-10, 22-11
- MIFLTD, 22-10
- MIFLTD status bit, 22-10
- MINST instruction, 22-16, 22-19
- misaligned destination register, 4-2
- miss penalties, TLB, 3-4
- MIX, 15-12
- MMU, 3-4, 5-5, 9-1-9-33
  - address, translation modes, 9-19
  - breakpoint control registers, 15-7
  - consistency, VBus, 18-9
  - control register, 9-22, 10-6, 14-25
  - control registers, 9-22
  - enable, 9-24
  - fault status register, 9-26

- flush operation, 8-8
- hit criteria, 9-11
- modified bits, 9-8
- operation, 5-10, 9-1-9-33
- probe, 9-12
- protection errors, 12-13, 12-15
- R&M updates, 10-35
- referenced bits, 9-8
- registers, 2-6, 9-22
- shadow FSR register, 9-35
- table walk, 10-12
- TLB, 9-36
- transparent mode, 9-18
- MMU.SB, 10-38
- models of memory, 8-2
- modified bit, 9-4, 9-8, 9-38
- module asynchronous error detect, 17-9
- module
  - ID, 17-15
  - identifier, 17-9
  - reset input, 17-9
- MOESI protocols, 17-2
- MPC/MNPC, 22-12
- MRESET, 22-10
- MSB, 4-5
- MSFSR register, 9-35, 22-10
- MSTAT, 21-7, 21-12, 21-13, 22-2, 22-8, 22-18, 22-19
  - register, 21-13, 22-8
  - scan chain, 21-9, 22-8
  - update operation, 22-10
- MSTAT.ERRMODE, 22-10
- MSTAT.MACK, 22-18
- MSTAT.MIDONE, 22-18
- MSTAT.MIFLTD, 22-18
- MSTAT.MIFLTD bit, 22-16
- MSTAT.PFPX, 22-25
- MSTAT.MIFLTD, 22-16
- MTMP[1-2], 22-12
- MTMP[1-2] registers, 22-12
- MTMP1 register, 22-21
- MULScc, 4-8
- multicache controller, 13-12, 14-23, 21-14
- multiple-instructions-per-cycle execution, 3-7
- multiply/divide register, register summary, 4-8
- multiprocessor
  - synchronization instructions, 2-2
  - multiprocessor system, 17-2

## N

- NaN, 4-14, 11-3
- NEW\_DATA\_VALUE, 22-23
- next program counter (nPC), 2-11, 4-9, 5-26, 22-6, 22-7
- NF bit, 9-20, 12-9
- no default
  - grantee, 19-47
  - bit, 9-24, 10-37, 12-9
  - operation, 9-20
- non-cacheable
  - accesses, 16-10
  - stores, 10-35
- non-emulation watchdog reset, 22-17
- non-maskable
  - interrupts, 12-11
  - requests, 12-11
- non-standard mode, 4-13, 11-6
- non-cacheable
  - loads, 8-7
  - stores, 8-7
- non-writable registers, 21-6
- NOPs, 22-6
- NS, 4-13
- numeric cases, iv, 11-1
- NWINDOWS, 2-3

## O

- odd destination register, 4-2
- odd-even pair of registers, 2-3
- on-chip PLL, 21-13
- one stall, 6-10
- operands, 2-2, 5-5
- operation, 5-1, 6-1
  - introduction, 5-2
- operation code, 16-43, 16-44
- oscilloscope, 15-15
- overall operation (processor), 2-3
- overflow
  - bit, 4-4
  - flag, 12-15
  - trap mask, 4-13
- overview of pipeline example, 5-7
- overwrite bit, 9-29, 9-34

## P

- PA
  - bit, 16-43
  - field, 16-43, 16-44
- packet detection, 19-16
- packet-switched
  - bus, 19-4
  - interface, 3-4
- packets, 19-15
- page table
  - entry, 9-4, 9-31
  - memory operations, 8-8
  - pointer, 3-4, 9-4, 9-5
  - hardware use of, 8-8
- PAMD, 15-9
- parallel daisy chains, 21-22
- parity, 12-13
  - enable, 9-24
  - error, 9-30
- partial store ordering, 3-5, 8-2, 9-24
- pass through/bypass transactions, 10-22
- PC, 4-7, 4-9, 5-2, 5-25, 22-6, 22-7
  - pair
    - altered non-emulation, 22-18*
    - captured non-emulation, 22-18*
  - value, 5-5, 5-23
- PCLK, 16-4, 17-42, 23-7
- PCLK and BCLK relationship, 23-7
- PEND\_, 8-11
- pending bit, 13-10
- performance, 6-2
- performance analysis, 3-5, 15-2
- periphery testing, 21-2
- PFPX, 22-11, 22-25
- phase locked loop, 23-1, 23-2
- physical address, 9-3, 16-43, 16-44
  - bits, 10-32
  - cache, 3-4
- physical page number, 9-3, 9-4, 9-37
- physical signal summary (MBus signals), 17-6
- physical value, 22-25
- PIL, 4-5, 15-13
- pin timings, 23-6
- pins, MXCC, D-2
- pipeline, 3-4, 5-5
  - bubble, 5-6

- bubbles, 5-4
  - example overview, 5-7
  - fundamentals, 5-4
  - hold cycle, 5-26
  - write-back, 5-6
- pipelined execution, 5-2
- PLL, 21-13, 23-1, 23-2
  - bypassed, 23-6
  - clock, 21-13, 23-3, 23-4
  - clock feedback loop, 23-2
  - enabled, 23-6
  - instability, 23-2
  - restabilization, 21-11
- port busy state, 17-29
- ports
  - integer register write, 6-4
  - to memory, 6-4
  - to the FPU, 6-4
- possible emulation instruction sequences, 22-19
- POST, 14-8
  - internal storage devices, 14-8
    - data cache, 14-12*
    - instruction cache, 14-11*
    - other POSTs, 14-13*
    - windowed register file, 14-8*
  - support routines, 14-13
    - data cache, 14-17*
    - instruction cache, 14-14*
- power-on reset, 10-6, 10-16, 14-6, 15-12
- power-on self-test, 14-8
- PPN, 9-3, 9-4, 9-37
- prefetch, 8-10
  - buffers, 13-7
  - exception handling, 8-10
  - exceptions, 10-12
- previous emulation instruction, 22-18
- primary
  - execution stage, 5-5
  - register, 21-6, 21-10, 21-13
  - register's scan chain, CID, 21-10
  - scan register, 21-10
- primitives, 22-19
- principles of operation, 5-1, 6-1
- prior non-emulation FPOPs, 22-11
- privilege error code, 9-32
- privileged
  - instructions, 2-3, 12-13
  - load/store alternate instructions, 2-6

- probe address, 9-12
- procedure call and return, 5-23
- processor
  - arbitration logic, 16-10
  - bus interface, 16-10
  - command logic, 16-9, 16-10
  - pipeline, 5-4
  - state register, 2-3, 4-3, 4-4, 22-19
  - status register, 6-21
- processor-dependent values, 2-6
- processor-state registers, 2-6
- program counter (PC), 2-3, 2-11, 4-7, 4-9, 5-2, 5-5, 12-7, 21-2
- program execution
  - block-stepping, 22-28
  - profile, 22-28
  - single-stepping, 22-28
- programmable timers, 15-2
- programs, self-modifying, 2-7
- protected registers, 2-6
- protocol commands, compound emulation, 22-17
- PS, 4-5
- pseudo-random test vectors, 21-11
- PSO, 3-5, 8-2, 8-3, 14-25
- PSR, 2-4, 4-4, 5-24, 6-21, 14-21, 22-21
- PSR.EC bit, 12-14
- PSR.EF, 12-14
- PSR.ET, 5-25, 12-10
- PSR.ET bit, 10-36, 12-11, 12-14
- PSR.IPL, 22-9
- PSR.PIL, 5-25, 12-11
- PSR.PS bit, 12-9
- PSR.S, 12-13
- Ptag format, 10-15, 10-30
- PTE, 9-4, 9-14, 9-16, 9-23, 9-31, 9-32, 9-33, 10-22
- PTP, 3-4, 9-4, 9-14, 9-31, 9-32
- ptp, 9-5

## Q

- quad precision and extended precision, FPU operation, 11-4
- quad-precision value, 2-3
- queue, 4-14

## R

- R&M update, 9-8

## R&R

- acknowledgement, 17-25, 17-27, 17-28, 17-29

- reply, 17-34, 17-40, 17-42, 17-47

ratio of PCLK to BCLK, 23-7

RD, 4-12

RDASR

RDPSR, 4-4, 4-5, 6-21

RD TBR instruction, 4-7

RDWIM instruction, 4-6

RDY, 4-8

read operands, 5-5

read valid command, 20-6

read/write control register, 2-5

reads

- double-word, 22-22

- half-word, 22-22

- signed, 22-22

- unsigned, 22-22

- word, 22-22

READY response, 17-40

receiver delay, 23-2

recovery mechanism, 21-2

reduction of branches, 6-7

reference clock, MBus, 17-13

reference/miss count register, 16-23

referenced bits, 9-8

register, 5-2, 9-22

- file, 13-7

- addresses, 2-2

- dual-port, 16-11

- floating-point, 2-2

- large-windowed, 2-2

- ports, 5-5, 6-19

- write port, 5-23

- global, integer unit, 2-3

- in, 2-3

- local, integer unit, 2-3

- out, 2-3

- pairs, 2-3

- state, 21-2

- summary, 4-1

- ancillary state register, 4-10

- floating-point *f* registers, 4-11

- floating-point queue, 4-16

- floating-point state registers, 4-12

- integer unit *r* registers, 4-2

- multiply/divide register, 4-8

- processor state register, 4-4

- program counters, 4-9*
  - trap base register, 4-7*
  - window invalid mask, 4-6*
- windows, 2-11, 4-2
- relinquish & retry reply, 17-34
- remote
  - debugging environment, 21-2
  - emulation, 21-9, 22-2, 22-18, 22-23,
- reply queue, 16-11
- request queue, 16-10, 16-11
- reset, 10-36, 14-2, 14-25
  - modes, 13-1
  - requirements, JTAG, 21-4
  - states, 14-21
    - caches, 14-22*
    - control registers, 14-21*
    - end reset, 14-27*
    - floating-point unit, 14-27*
    - MFSR, 14-22*
    - MMU control register, 14-25*
    - multicache controller, 14-23*
    - PSR, 14-21*
    - superscalar execution, 14-21*
    - TLB, 14-22*
  - trap, 12-13, 13-6
  - types, 13-2, 13-4
- resource allocation, 5-5, 6-4
- RESTORE, 4-4, 4-5, 4-6, 5-23, 12-14, 22-21
- retry acknowledgement, 17-27
- RETT, 4-5, 4-6, 5-26, 6-22
  - instruction, 4-4, 4-6, 12-9, 12-14
  - pipeline, 5-26
- return from trap pipeline, 5-26
- ring size, 21-6
- RISC, 3-4, 5-8
- root cache, 9-10
- root pointer cache, 3-4
- rounding operations, FPU operation, 11-4
- routing delay, internal, 23-2
- Run-for-N Instructional Cycles, 22-28

## S

- S bit, 4-5, 16-43, 16-44
- SAVE instruction, 4-4, 4-5, 4-6, 5-23, 12-14
- SBCNTL.Dptr, 10-39
- SBTAGS.SP, 7-6



- scan chain, 21-6, 21-10
  - CID primary register's, 21-10
  - JTAG, 21-6
  - registers, primary, 21-6
- scan rings, independent, 21-12
- SDIV instruction, 4-8
- SDIVcc instruction, 4-8
- second emulation session, 22-18
- second operand, 12-16
- second-level
  - JTAG controller, 21-22
  - TAP controller, 21-22
- SEE\_PLL, 21-13
- select, 9-36
- self test, 13-7
- self-aligning nature of execution, 6-7
- separate I&D memories, 2-7
- sequence errors, 12-15
- serial scan chains, JTAG-accessible, 21-7
- SETHI, SPARC architecture, instructions, 2-7
- setting code and data address breakpoints, 22-25
- shadow FSR, 22-10
- shared
  - bit, 10-32
  - signal, 17-14
  - write, VBus transactions and waveforms, 18-32
- SHIFT, 2-7, 21-5, 21-6, 21-8
- shift register, 21-6
- shifter, 5-6
- SHORT\_BIST, 13-7, 21-7
- SI reset, 13-10
- signal user emulation request (SIGM), 4-10, 7-7, 22-14
- signature
  - register, 21-7, 21-11
  - scan chain, 21-11
  - value, 13-7
- SIGNATURE TDR, 21-11
- signed and unsigned half-word and word and double-word reads, 22-22
- signed
  - byte, 22-22
  - reads, 22-22
- single registers, 2-3
- single- and double-precision floating-point arithmetic functions, high-performance, 3-4
- single-chip interface, 3-4
- single-point memory references, 6-14
- single-precision

- operand, 4-11
- value, 2-3
- single-word wide, 22-25
- SIZE, 17-44
- slave, 17-2
- SMUL, 4-8
- SMULcc, 4-8
- snoop
  - enable, 9-23
  - hit, 5-6, 10-10, 10-24, 10-27
  - on the bus, 5-6
- snoop-enable bit, 10-13
- snooping
  - bus, 3-5
  - caches, 17-25
- software
  - debugging facilities, 15-2
  - internal reset, 13-10
    - MXCC, 13-11
  - memory-scrubbing scheme, hardware-assisted, 16-30
- software-controlled boundary scan, 21-2
- source
  - address, 16-30
  - register, 6-13
- space atomics (alternate), 8-5
- SPARC
  - architecture
    - coprocessor, 2-4*
    - floating-point unit, 2-3*
    - input/output, 2-10*
    - instructions, 2-5*
      - arithmetic, 2-7
      - control transfer, 2-7
      - coprocessor operate, 2-8
      - floating-point operate, 2-8
      - load/store, 2-5
      - logical, 2-7
      - memory access, 2-5
      - SETHI, 2-7
      - shift, 2-7
      - state register access, 2-8
    - integer unit, 2-3*
    - introduction, 2-2*
    - manual, 2-1*
    - memory model, 2-9*
      - partial store ordering (PSO), 2-9
      - total store ordering (TSO), 2-9
    - SPARC reference MMU, 2-13*
    - traps, 2-11*
    - summary, 2-1*

- assembly language, 21-2
- implementation, 2-3
- instruction, 22-16
- instruction set, 22-16
- International, Inc., 2-2
- processor, 2-2
- reference memory management unit (MMU), 2-13, 3-4
  - control registers*, 9-22
  - operation*, 9-23
  - specification*, iii, 9-1
- registers, 10-34
- split
  - after
    - any control transfer instruction*, 6-17
    - condition codes set in cascade rule*, 6-17
    - first instruction after anulled branch rule*, 6-17
    - first valid exception rule*, 6-16
    - MULSCC destination not equal to source of next MULSCC rule*, 6-17
  - before
    - cascade into*
      - JMPL rule, 6-20
      - memory reference address, 6-20
      - shift rule, 6-20
    - control register read after previous SetCC rule*, 6-21
    - delay group CTI unless first rule*, 6-22
    - extended arithmetic from CC set in current group*, 6-22
    - invalid instruction rule*, 6-19
    - load data cascade use rule*, 6-20
    - MULSCC unless first one or two instructions rule*, 6-22
    - out of integer register ports rule*, 6-19
    - previous group cascade into memory reference address rule*, 6-20
    - second cascade rule*, 6-19
    - second shift rule*, 6-19
    - sequential instruction rule*, 6-21
  - between add and shift, 5-9
- spread address calculation, 6-14
- square root, 5-14
- squashed instruction, 5-25-21
- SRAM chips, 16-4
- ST instruction, 3-4, 5-6, 6-10
- STA emulation instruction, 8-6, 10-35, 13-2, 13-7, 22-10
- stable clock sources, 23-6
- Stag format, 10-15, 10-30
- stages, 5-4
- stale data, 17-14
- startup procedure, 14-1
  - POST, 14-8
  - power-on self-test, 14-8
  - reset handling, 14-2

- reset states, 14-21
  - caches, 14-22*
  - control registers, 14-21*
  - end reset, 14-27*
  - floating-point unit, 14-27*
  - MFSR, 14-22*
  - MMU control register, 14-25*
  - multicache controller, 14-23*
  - PSR, 14-21*
  - superscalar execution, 14-21*
  - TLB, 14-22*
- state
  - during emulation mode, 22-15*
  - register access
    - ancillary state, instructions, SPARC architecture, 2-8*
    - instructions, SPARC architecture, 2-8*
  - vectors, signature, 21-9*
- STBAR instruction, 4-10, 7-6, 10-41
- STDFQ instruction, 4-14, 4-16, 5-16, 22-25
- STEN\_CBK, 15-13
- STEN\_DBK, 15-13
- STEN\_ZCC, 15-13
- STEN\_ZIC, 15-13
- STFSR instruction, 4-12, 4-14
- store, 22-21
  - address, 5-10*
  - alternate, 8-6, 10-35*
  - barrier (STBAR), 7-6, 10-41*
  - buffer, 8-7, 9-8, 10-26, 10-34, 13-5, 13-7*
    - data store exceptions, 10-36*
    - diagnostics and control, 10-40*
    - disabled operation—strong ordering, 10-39*
    - general operation, 10-34*
    - high integration, Viking introduction, 3-4*
    - non-buffered operations, 10-35*
    - synchronous operations, 10-35*
  - control, 10-38*
  - control register, 10-42*
  - copy-out, 5-12, 8-4*
  - data, 10-38*
  - data register, 10-41*
  - depth, 12-11*
  - disabled, 10-42*
  - empty bit, 10-42*
  - enable, 9-24*
  - buffer enabled, 10-42*
  - error, 9-30*
  - error pending bit, 10-42*
  - errors, 9-27*

- non-empty*, 10-42
  - snooping*, 5-12
  - tag*, 10-38
  - tag register*, 10-37, 14-25
  - tags register*, 10-40
- FSR operations, 12-15, 22-25
- instructions, 3-4
- stream
  - data
    - buffer*, 16-30
    - register*, 16-31
  - destination register, 16-32
  - source register, 16-30, 16-31
  - stream write operations, 16-32
- strong
  - consistency, 2-9
  - ordering, 8-2
- STSFR instruction, 4-14
- style and symbol conventions, vii
- sub-block
  - address, 16-30
  - states, 19-21
- Sun Microsystems, Inc., 2-2
- superscalar
  - design, 3-3
  - execution, 5-2, 13-5, 14-21
- SuperSPARC
  - configurations, 1-4, 17-2
  - introduction, 1-1, 1-2
  - processor, 21-7
- supervisor
  - access indicator, 17-10
  - bit, 10-41
  - data space, 22-22
  - mode, 2-3
  - software, trap, 2-3
  - state, 16-43, 16-44, 17-10
- supervisor-only instructions, 2-3
- supervisor-only page, 9-32
- SWAP, 15-3
  - access, 13-8
  - instruction, 8-4, 17-34, 17-42
  - transactions, 9-8
- SWAPA instruction, B-1
- symbolic constants, 22-20
- synchronous
  - external stores, 12-12
  - operation, 23-7
  - SRAM, 16-4

## system

- clock skew, 23-1
  - configurations, SuperSPARC, 1-4
  - noise, 23-3, 23-4
  - reset, 13-10
  - software use of page tables, 8-8
- system implementation-dependent error, 17-31, 17-32, 19-19, 19-20
- system-dependent values, 2-6
- system-level test, 21-22

## T

- table walk cacheable bit, 9-14, 9-23, 10-12
- TADDCCTV instruction, 12-15
- tag
- commands, 19-32
  - comparison, 5-5
- tagged
- add, 12-15
  - data instructions, 2-2
  - operation, 4-4
  - operation overflow trap, 12-15
  - subtract, 12-15
- taken branch, 5-19, 5-20, 5-21, 5-23, 6-2, 6-7, 6-17
- TAP
- controller, 21-4, 21-6, 23-2
    - JTAG, 21-4, 21-11, 23-2
    - second-level, 21-22
    - reset, 22-8
    - state machine, 21-3
  - reset state, 13-2, 21-11
- TBA, 4-7
- TBR, 4-7, 22-21
- TC, 9-23
- TCK cycle, 13-2, 21-3, 21-4, 21-6, 21-10, 21-11, 21-22, 22-8, 23-2
- TDI, 21-3, 21-10, 21-13, 21-22
- data, 21-10
  - value, 21-6
- TDO, 21-3, 21-6, 21-10, 21-13, 21-22
- TDR scan chain, 21-8
- TEM bit, 4-12, 4-15
- TEM value, 4-12
- TEM.DZM, 4-13
- TEM.NVM, 4-13
- TEM.NXM, 4-13
- TEM.OFM, 4-13
- TEM.UFM, 4-13

- test
  - access point controller, 21-4
  - clock, 21-3, 22-8, 23-2
  - data
    - in*, 21-3
    - out*, 21-3
  - mode select, 21-3
  - vectors, pseudo-random, 21-11
- TEST LOGIC RESET state, 21-3, 21-4
- JTAG, 21-4
- testability, high integration, Viking introduction, 3-6
- third-level JTAG controller, 21-22
- three models of memory, 8-2
- three-stated, 20-6
- throughput, 6-10
- TICC instructions, 4-4, 4-7, 12-16
- time-out, 9-30
  - error, 16-13
  - response, 16-13, 17-39, 17-47
- timing summary, MBus, 17-48
- TLB, 5-5, 9-3, 9-18, 9-36, 10-12, 13-5, 13-7, 14-22
  - entry, 9-9, 9-36
  - miss penalties, 3-4
  - replacement policy, 9-9
- TMRM, 22-8, 22-9
- TMS, 13-2, 21-3, 21-4, 21-11, 21-22 pin, 23-2
- total store ordering, 3-5, 8-2, 9-24
- transactions, 19-16
  - and waveforms, VBus, 18-17
- transfer, 16-30
- transient requests, 12-11
- translation look aside buffer, 9-36
- transparent mode, 9-18
- trap, 2-5, 12-1
  - and the store buffer, 12-10
  - base
    - address*, 4-7
    - register (TBR)*, 2-11, 4-7
  - deferred, 2-11
  - categories, 2-11
  - details, 12-13
  - exception mask, 4-12
  - handler, 2-11
  - handling, 2-2
  - instructions, 12-16
  - interrupting, 2-11
  - precise, 2-11
  - priorities, 12-8

- type, 2-11
  - SPARC architecture, 2-11
  - to supervisor software, 2-3
- type, 12-4
  - field, 4-7, 12-9
  - not implemented, 12-6
- trapped floating-point instruction, 2-4
- tri-state, 20-6
- TSO, 3-5, 8-2
- TSUBCCTV instruction, 12-15
- TT field, 4-7, 12-9
- two-byte boundaries, 2-5
- type field, 9-12

## U

- u, 4-12
- UD/UC/TO/BE errors, 9-30
- UDIV instruction, 4-8
- UDIVcc instruction, 4-8
- UMUL, 4-8
- UMULcc, 4-8
- unassigned opcode, 12-14
- uncorrectable
  - error, 9-30
  - response, 17-39, 17-47
- undefined error, 9-30
- underflow
  - detection, FPU operations, 11-4
  - trap mask, 4-13
- unfinished FPop exceptions, FPU operation, 11-3
- unimplemented instruction trap, 12-14
- uniprocessor system, 17-2
- unordered relation, 4-14
- unsigned reads, 22-22
- untaken branch, 5-19, 5-21, 6-17
- UPDATE operation, 21-5, 21-6, 21-8, 21-10, 21-11, 21-13
- user mode, 2-3
  - SPARC architecture, 2-3
- user-application program, 2-3
- user-application programs, 2-3

## V

- valid bit, 10-41, 13-10
- VBus, 18-1
  - arbitration, 18-8





1. The first part of the document is a list of names and addresses, which appears to be a directory or a list of contacts. The names are written in a cursive script, and the addresses are listed below them.

2. The second part of the document is a list of names and addresses, which appears to be a directory or a list of contacts. The names are written in a cursive script, and the addresses are listed below them.

3. The third part of the document is a list of names and addresses, which appears to be a directory or a list of contacts. The names are written in a cursive script, and the addresses are listed below them.

4. The fourth part of the document is a list of names and addresses, which appears to be a directory or a list of contacts. The names are written in a cursive script, and the addresses are listed below them.

5. The fifth part of the document is a list of names and addresses, which appears to be a directory or a list of contacts. The names are written in a cursive script, and the addresses are listed below them.

6. The sixth part of the document is a list of names and addresses, which appears to be a directory or a list of contacts. The names are written in a cursive script, and the addresses are listed below them.

7. The seventh part of the document is a list of names and addresses, which appears to be a directory or a list of contacts. The names are written in a cursive script, and the addresses are listed below them.

8. The eighth part of the document is a list of names and addresses, which appears to be a directory or a list of contacts. The names are written in a cursive script, and the addresses are listed below them.

9. The ninth part of the document is a list of names and addresses, which appears to be a directory or a list of contacts. The names are written in a cursive script, and the addresses are listed below them.

10. The tenth part of the document is a list of names and addresses, which appears to be a directory or a list of contacts. The names are written in a cursive script, and the addresses are listed below them.

1. The first step is to identify the problem or question that needs to be answered. This involves understanding the context and the specific requirements of the task.

- window
  - address, 4-3
  - invalid mask, register summary, 2-2, 4-6
  - overflow, 4-6, 12-14
  - window underflow trap, 12-14
- window\_overflow trap, 4-6
- window\_underflow trap, 4-6
- windowed register file, 14-8
- word, 2-5
  - access, 2-5
- word
  - reads, 22-22
  - word reference, 5-8, 12-15
- wrapping, 17-15
- write
  - hits in stream transfers, 19-40
  - invalidate protocol, 17-14
  - memory, 22-22
  - valid command, 20-6
- write-allocate, 5-6
- write-back policy, 17-13
- write-invalidate cache-consistency protocol, 17-25
- write-through cache, 5-6
- write-through mode, 10-21
- WRPSR instruction, 4-4, 4-5, 4-7, 12-14
- WRTBR instruction, 4-7
- WRWIM instruction, 4-6
- WRY, 4-8

## X

- XADDR, 22-22, 22-23
- XBus, 3-4, 19-1
  - arbiter, 16-11
  - arbitration priorities, 19-44
  - bus commands, 19-23
    - data commands, 19-23*
    - tag commands, 19-32*
  - bus cycle waveforms, 19-49
  - bus ownership, 19-46
    - default grantee, 19-47*
    - no default grantee, 19-47*
  - bus protocol, 19-15
    - cycles, 19-15*
    - packet detection, 19-16*
    - packets, 19-15*
    - transactions, 19-16*
  - cache consistency protocols, 19-21
    - sub-block states, 19-21*

- XBus configuration, 19-21*
- configuration, 17-5, 19-21
- default grantee, 19-46
- flow control, 19-45
  - bus watchers, 19-46*
- message priority detection, 19-48
- overview, 19-2
  - bus watchers, 19-6*
  - circuit-switched bus, 19-4*
  - operation, 19-9*
  - packet-switched bus, 19-4*
- write hits in stream transfers, 19-40
- XREQ (overbar) decoding, 19-41
- XBus/MBus interface, 16-10
- XHPC, 12-7
- XNPC, 12-7
- XPC, 12-7
- XREG register, 22-20, 22-23
- XREQ (overbar) decoding, 19-41

## **Y**

- Y register, 4-8, 22-21

## **Z**

- ZCCIS, 15-12
- ZCCM, 22-8, 22-9
- zero cycle count breakpoint, 22-9
- zero-instruction-count breakpoint, 22-9
- ZICIS, 15-12
- ZICM, 22-8, 22-9